JDBC AND DATABASE CONNECTION POOLING

Topics in This Chapter

• The seven basic steps in connecting to databases
• Simple database retrieval example
• Some utilities that simplify JDBC usage
• Formatting a database result as plain text or HTML
• An interactive graphical query viewer
• Precompiled queries
• A connection pool library
• A comparison of servlet performance with and without connection pooling
• Sharing connection pools

Online version of this first edition of Core Servlets and JavaServer Pages is free for personal use. For more information, please see:
• Second edition of the book:
• Sequel:
• Servlet and JSP training courses from the author:
DBC provides a standard library for accessing relational databases. Using the JDBC API, you can access a wide variety of different SQL databases with exactly the same Java syntax. It is important to note that although JDBC standardizes the mechanism for connecting to databases, the syntax for sending queries and committing transactions, and the data structure representing the result, it 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 in your code.

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.
Chapter 18  JDBC and Database Connection Pooling

Core Note

JDBC is not an acronym.

Although a complete tutorial on database programming is beyond the scope of this chapter, I’ll cover the basics of using JDBC here, assuming you are already familiar with SQL. For more details on JDBC, see http://java.sun.com/products/jdbc/, the on-line API for java.sql, or the JDBC tutorial at http://java.sun.com/docs/books/tutorial/jdbc/. If you don’t already have access to a database, you might find mySQL a good choice for practice. It is free for non-Microsoft operating systems as well as for educational or research use on Windows. For details, see http://www.mysql.com/.

18.1 Basic Steps in Using JDBC

There are seven standard steps in querying databases:

1. Load the JDBC driver.
2. Define the connection URL.
3. Establish the connection.
4. Create a statement object.
5. Execute a query or update.
6. Process the results.
7. Close the connection.

Here are some details of the process.

Load the Driver

The driver is the piece of software that knows how to talk to the actual database server. To load the driver, all you need to do is to load the appropriate class; a static block in the class itself automatically makes a driver instance and registers it with the JDBC driver manager. To make your code as flexible as possible, it is best to avoid hard-coding the reference to the class name.

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: use Class.forName. This method takes a string represent-
ing a fully qualified class name (i.e., one that includes package names) and loads the corresponding class. This call could throw a `ClassNotFoundException`, so should be inside a `try/catch` block. Here is an example:

```java
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 specific format required by the server. This approach means that you have to obtain a JDBC driver specific to the database you are using; you will need to check its documentation for the fully qualified class name to use. Most database vendors supply free JDBC drivers for their databases, but there are many third-party vendors of drivers for older databases. For an up-to-date list, see [http://java.sun.com/products/jdbc/drivers.html](http://java.sun.com/products/jdbc/drivers.html). Many of these driver vendors supply free trial versions (usually with an expiration date or with some limitations on the number of simultaneous connections), so it is easy to learn JDBC without paying for a driver.

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, be sure to include the path to the JAR file in your CLASSPATH setting.

### Define the Connection URL

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

```java
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;
```
JDBC is most often used from servlets or regular desktop applications but is also sometimes employed from applets. If you use JDBC from an applet, remember that, to prevent hostile applets from browsing behind corporate firewalls, browsers prevent applets from making network connections anywhere except to the server from which they were loaded. Consequently, to use JDBC from applets, either the database server needs to reside on the same machine as the HTTP server or you need to use a proxy server that reroutes database requests to the actual server.

**Establish the Connection**

To make the actual network connection, pass the URL, the database username, and the 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. I'm 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.

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

An optional part of this step is to look up information about the database by using the `getMetaData` method of `Connection`. This method returns a `DatabaseMetaData` object which has methods to let you discover the name and version of the database itself (`getDatabaseProductName`, `getDatabaseProductVersion`) or of the JDBC driver (`getDriverName`, `getDriverVersion`). Here is an example:

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

Other useful methods in the `Connection` class include `prepareStatement` (create a `PreparedStatement`; discussed in Section 18.6), `prepareCall` (create a `CallableStatement`), `rollback` (undo statements since last commit), `commit` (finalize operations since last commit), `close` (terminate connection), and `isClosed` (has the connection either timed out or been explicitly closed?).
Create a Statement

A `Statement` object is used to send queries and commands to the database and is created from the `Connection` as follows:

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

Execute a Query

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:

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

To modify the database, use `executeUpdate` instead of `executeQuery`, and supply a string that uses `UPDATE`, `INSERT`, or `DELETE`. Other useful methods in the `Statement` class include `execute` (execute an arbitrary command) and `setQueryTimeout` (set a maximum delay to wait for results). You can also create parameterized queries where values are supplied to a precompiled fixed-format query. See Section 18.6 for details.

Process the Results

The simplest way to handle the results is to process them one row at a time, using the `ResultSet`'s `next` method to move through the table a row at a time. Within a row, `ResultSet` provides various `getXxx` methods that take a column index or column name as an argument and return the result as 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` regardless of the actual column type. However, if you use the version that takes a column index, 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 three columns in all rows of a `ResultSet`. 

Chapter 18  JDBC and Database Connection Pooling

while(resultSet.next()) {
    System.out.println(results.getString(1) + " " +
                      results.getString(2) + " " +
                      results.getString(3));
}

In addition to the `getXxx` and `next` methods, other useful methods in the `ResultSet` class include `findColumn` (get the index of the named column), `wasNull` (was the last `getXxx` result SQL NULL? Alternatively, for strings you can simply compare the return value to null), and `getMetaData` (retrieve information about the `ResultSet` in a `ResultSetMetaData` object).

The `getMetaData` method is particularly useful. Given only a `ResultSet`, you have to know about 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 include `getColumnCount` (the number of columns), `getColumnIndex` (a column name, indexed starting at 1), `getColumnType` (an int to compare against entries in `java.sql.Types`), `isReadOnly` (is entry a read-only value?), `isSearchable` (can it be used in a `WHERE` clause?), `isNullable` (is a null value permitted?), and several others that give details on the type and precision of the column. `ResultSetMetaData` does not include the number of rows, however; the only way to determine that is to repeatedly call `next` on the `ResultSet` until it returns `false`.

**Close the Connection**

To close the connection explicitly, you would do:

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

You should postpone this step 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 Section 18.7 develops a library just for that purpose and Section 18.8 shows some typical timing results.
18.2 Basic JDBC Example

Listing 18.3 presents a simple class called FruitTest that follows the seven steps outlined in the previous section to show a simple table called fruits. It uses the command-line arguments to determine the host, port, database name, and driver type to use, as shown in Listings 18.1 and 18.2. Rather than putting the driver name and the logic for generating an appropriately formatted database URL directly in this class, these two tasks are spun off to a separate class called DriverUtilities, shown in Listing 18.4. This separation minimizes the places that changes have to be made when different drivers are used.

This example does not depend on the way in which the database table was actually created, only on its resultant format. So, for example, an interactive database tool could have been used. In fact, however, JDBC was also used to create the tables, as shown in Listing 18.5. For now, just skim quickly over this listing, as it makes use of utilities not discussed until the next section.

Also, a quick reminder to those who are not familiar with packages. Since FruitTest is in the coreservlets package, it resides in a subdirectory called coreservlets. Before compiling the file, I set my CLASSPATH to include the directory containing the coreservlets directory (the JAR file containing the JDBC drivers should be in the CLASSPATH also, of course). With this setup, I compile simply by doing “javac FruitTest.java” from within the coreservlets subdirectory. But to run FruitTest, I need to refer to the full package name with “java coreservlets.FruitTest ...”.

Listing 18.1  FruitTest result (connecting to Oracle on Solaris)

Prompt> java coreservlets.FruitTest dbhost1.apl.jhu.edu PTE hall xxxx oracle
Database: Oracle
Version: Oracle7 Server Release 7.2.3.0.0 - Production Release
PL/SQL Release 2.2.3.0.0 - Production

Comparing Apples and Oranges
====================================
QUARTER | APPLES | APPLESALES | ORANGES | ORANGESALES | TOPSELLER
1        | 32248  | $3547.28   | 18459   | $3138.03    | Maria
2        | 35009  | $3850.99   | 18722   | $3182.74    | Bob
3        | 39393  | $4333.23   | 18999   | $3229.83    | Joe
4        | 42001  | $4620.11   | 19333   | $3286.61    | Maria
Chapter 18  JDBC and Database Connection Pooling

Listing 18.2  FruitTest result (connecting to Sybase on NT)

Prompt> java coreservlets.FruitTest dbhost2.apl.jhu.edu 605741 hall xxxx sybase
Database: Adaptive Server Anywhere
Version: 6.0.2.2188
Comparing Apples and Oranges
-------------------------------
quarter   apples   applesales   oranges   orangesales   topseller
 1    32248     $3547.28       18459     $3138.03      Maria
 2    35009     $3850.99       18722     $3182.74      Bob
 3    39393     $4333.23       18999     $3229.83      Joe
 4    42001     $4620.11       19333     $3286.61      Maria

Listing 18.3  FruitTest.java

package coreservlets;

import java.sql.*;

/** A JDBC example that connects to either an Oracle or
 *  a Sybase database and prints out the values of
 *  predetermined columns in the "fruits" table.
 */

public class FruitTest {

/** Reads the hostname, database name, username, password,
 *  and vendor identifier from the command line. It
 *  uses the vendor identifier to determine which
 *  driver to load and how to format the URL. The
 *  driver, URL, username, host, and password are then
 *  passed to the showFruitTable method.
 */

public static void main(String[] args) {
    if (args.length < 5) {
        printUsage();
        return;
    }
    String vendorName = args[4];
    int vendor = DriverUtilities.getVendor(vendorName);
    if (vendor == DriverUtilities.UNKNOWN) {
        printUsage();
        return;
    }
18.2 Basic JDBC Example

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];
showFruitTable(driver, url, username, password);
}

/** Get the table and print all the values. */
public static void showFruitTable(String driver,
        String url,
        String username,
        String password) {
try {
    // Load database driver if 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();
    String productName =
            dbMetaData.getDatabaseProductName();
    System.out.println("Database: " + productName);
    String productVersion =
            dbMetaData.getDatabaseProductVersion();
    System.out.println("Version: " + productVersion + "\n");
    System.out.println("Comparing Apples and Oranges\n" +
            "============================");
    Statement statement = connection.createStatement();
    String query = "SELECT * FROM fruits";
    // Send query to database and store results.
    ResultSet resultSet = statement.executeQuery(query);
    // Look up information about a particular table.
    ResultSetMetaData resultsMetaData =
            resultSet.getMetaData();
    int columnCount = resultsMetaData.getColumnCount();
    // Column index starts at 1 (a la SQL) not 0 (a la Java).
    for(int i=1; i<columnCount+1; i++) {
        System.out.print(resultsMetaData.getColumnName(i) +
                "  ");
    }
    System.out.println();
    // Print results.
    while(resultSet.next()) {

Listing 18.3 FruitTest.java (continued)
Listing 18.3  FruitTest.java (continued)

```java
// Quarter System.out.print("" + resultSet.getInt(1));
// Number of Apples System.out.print("" + resultSet.getInt(2));
// Apple Sales System.out.print("" + resultSet.getFloat(3));
// Number of Oranges System.out.print("" + resultSet.getInt(4));
// Orange Sales System.out.print("" + resultSet.getFloat(5));
// Top Salesman System.out.println("" + resultSet.getString(6));
} catch(ClassNotFoundException cnfe) {
    System.err.println("Error loading driver: " + cnfe);
} catch(SQLException sqle) {
    System.err.println("Error connecting: " + sqle);
}
}

private static void printUsage() {
    System.out.println("Usage: FruitTest host dbName " +
            "username password oracle|sybase.");
}
```

Listing 18.4  DriverUtilities.java

```java
package coreservlets;

/** Some simple utilities for building Oracle and Sybase * JDBC connections. This is <I>not</I> general-purpose * code -- it is specific to my local setup. */

public class DriverUtilities {
    public static final int ORACLE = 1;
    public static final int SYBASE = 2;
    public static final int UNKNOWN = -1;
```
Listing 18.4  DriverUtilities.java (continued)

/** Build a URL in the format needed by the
 * Oracle and Sybase drivers I am using. */

public static String makeURL(String host, String dbName,
  int vendor) {
  if (vendor == ORACLE) {
    return("jdbc:oracle:thin:@" + host + ":1521:" + dbName);
  } else if (vendor == SYBASE) {
    return("jdbc:sybase:Tds:" + host + ":1521" +
           "?SERVICENAME=" + dbName);
  } else {
    return(null);
  }
}

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

public static String getDriver(int vendor) {
  if (vendor == ORACLE) {
    return("oracle.jdbc.driver.OracleDriver");
  } else if (vendor == SYBASE) {
    return("com.sybase.jdbc.SybDriver");
  } else {
    return(null);
  }
}

/** Map name to int value. */

public static int getVendor(String vendorName) {
  if (vendorName.equalsIgnoreCase("oracle")) {
    return(ORACLE);
  } else if (vendorName.equalsIgnoreCase("sybase")) {
    return(SYBASE);
  } else {
    return(UNKNOWN);
  }
}
Listing 18.5 FruitCreation.java

package coreservlets;
import java.sql.*;

/** Creates a simple table named "fruits" in either
 * an Oracle or a Sybase database.
 */
public class FruitCreation {
    public static void main(String[] args) {
        if (args.length < 5) {
            printUsage();
            return;
        }
        String vendorName = args[4];
        int vendor = DriverUtilities.getVendor(vendorName);
        if (vendor == DriverUtilities.UNKNOWN) {
            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];
        String format = "(quarter int, " +
                        "apples int, applesales float, " +
                        "oranges int, orangesales float, " +
                        "topseller varchar(16))";
        String[] rows = {
                         "(1, 32248, 3547.28, 18459, 3138.03, 'Maria')",
                         "(2, 35009, 3850.99, 18722, 3182.74, 'Bob')",
                         "(3, 39393, 4333.23, 18999, 3229.83, 'Joe')",
                         "(4, 42001, 4620.11, 19333, 3286.61, 'Maria')" }
        Connection connection = DatabaseUtilities.createTable(driver, url,
                                                               username, password,
                                                               "fruits", format, rows,
                                                               false);
        // Test to verify table was created properly. Reuse
        // old connection for efficiency.
        DatabaseUtilities.printTable(connection, "fruits",
                                       11, true);
    }
    private static void printUsage() {
        System.out.println("Usage: FruitCreation host dbName " +
                           "username password oracle|sybase.");
    }
}
18.3 Some JDBC Utilities

In many applications, you don’t need to process query results a row at a time. For example, in servlets and JSP pages, it is common to simply format the database results (treating all values as strings) and present them to the user in an HTML table (see Sections 18.4 and 18.8), in an Excel spreadsheet (see Section 11.2), or distributed throughout the page. In such a case, it simplifies processing to have methods that retrieve and store an entire ResultSet for later display.

This section presents two classes that provide this basic functionality along with a few formatting, display, and table creation utilities. The core class is DatabaseUtilities, which implements static methods for four common tasks:

1. **getQueryResults**
   This method connects to a database, executes a query, retrieves all the rows as arrays of strings, and puts them inside a DBResults object (see Listing 18.7). This method also places the database product name, database version, the names of all the columns and the Connection object into the DBResults object. There are two versions of getQueryResults: one that makes a new connection and another that uses an existing connection.

2. **createTable**
   Given a table name, a string denoting the column formats, and an array of strings denoting the row values, this method connects to a database, removes any existing versions of the designated table, issues a CREATE TABLE command with the designated format, then sends a series of INSERT INTO commands for each of the rows. Again, there are two versions: one that makes a new connection and another that uses an existing connection.

3. **printTable**
   Given a table name, this method connects to the specified database, retrieves all the rows, and prints them on the standard output. It retrieves the results by turning the table name into a query of the form “SELECT * FROM tableName” and passing it to getQueryResults.
4. **printTableData**

Given a `DBResults` object from a previous query, this method prints it on the standard output. This is the underlying method used by `printTable`, but it is also useful for debugging arbitrary database results.

Listing 18.6 gives the main code, and Listing 18.7 presents the auxiliary `DBResults` class that stores the accumulated results and returns them as arrays of strings (`getRow`) or wrapped up inside an HTML table (`toHTMLTable`). For example, the following two statements perform a database query, retrieve the results, and format them inside an HTML table that uses the column names as headings with a cyan background color.

```java
DBResults results =
    DatabaseUtilities.executeQueryResults(driver, url,
        username, password,
        query, true);
out.println(results.toHTMLTable("CYAN"));
```

Since an HTML table can do double duty as an Excel spreadsheet (see Section 11.2), the `toHTMLTable` method provides an extremely simple method for building tables or spreadsheets from database results.

Remember that the source code for `DatabaseUtilities` and `DBResults`, like all the source code in the book, can be downloaded from www.coreservlets.com and used or adapted without restriction.

---

**Listing 18.6 DatabaseUtilities.java**

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

public class DatabaseUtilities {

/** Connect to database, execute specified query,
 * and accumulate results into DBResults object.
 * If the database connection is left open (use the
 * close argument to specify), you can retrieve the
 * connection with DBResults.getConnection.
 */

public static DBResults getQueryResults(String driver,
    String url,
    String username, String password,
    String query, boolean close) {
```
18.3 Some JDBC Utilities

**Listing 18.6 DatabaseUtilities.java (continued)**

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

/** Retrieves results as in previous method but uses
 * an existing connection instead of opening a new one.
 */

public static DBResults getQueryResults(Connection connection,
                                        String query,
                                        boolean close) {
    try {
        DatabaseMetaData dbMetaData = connection.getMetaData();
        String productName =
            dbMetaData.getDatabaseProductName();
        String productVersion =
            dbMetaData.getDatabaseProductVersion();
        Statement statement = connection.createStatement();
        ResultSet resultSet = statement.executeQuery(query);
        ResultSetMetaData resultsMetaData =
            resultSet.getMetaData();
        int columnCount = resultsMetaData.getColumnCount();
        String[] columnNames = new String[columnCount];
        // Column index starts at 1 (a la SQL) not 0 (a la Java).
        for(int i=1; i<columnCount+1; i++) {
            columnNames[i-1] =
                resultsMetaData.getColumnName(i).trim();
        }
        DBResults dbResults =
            new DBResults(connection, productName, productVersion,
                          columnCount, columnNames);
        while(resultSet.next()) {
            String[] row = new String[columnCount];
            // Again, ResultSet index starts at 1, not 0.
            for(int i=1; i<columnCount+1; i++) {
                String entry = resultSet.getString(i);
                if (entry != null) {
                    entry = entry.trim();
                }
            }
        }
    } finally {
        if (close) {
            try {
                // Clean up connections and other resources.
            } finally {
                // Print summary of work done.
            }
        }
    }
    return(dbResults);
}
```
Listing 18.6 DatabaseUtilities.java (continued)

    row[i-1] = entry;
    } 
    dbResults.addRow(row);
    }
    if (close) {
        connection.close();
    }
    return(dbResults);
}
} catch(SQLException sqle) {
    System.err.println("Error connecting: " + sqle);
    return(null);
}
}

/** Build a table with the specified format and rows. */

public static Connection createTable(String driver,
    String url,
    String username,
    String password,
    String tableName,
    String tableFormat,
    String[] tableRows,
    boolean close) {

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

/** Like the previous method, but uses existing connection. */

public static Connection createTable(Connection connection,
    String username,
    String password,
    String tableName,
    String tableFormat,
    String[] tableRows,
    boolean close) {

try {

    Statement statement = connection.createStatement();
    // Drop previous table if it exists, but don’t get
    // error if it doesn’t. Thus the separate try/catch here.
    try {
        statement.execute("DROP TABLE " + tableName);
    } catch(SQLException sqle) {

    }
    String createCommand = "CREATE TABLE " + tableName + " " + tableFormat;
    statement.execute(createCommand);
    String insertPrefix = "INSERT INTO " + tableName + " VALUES";
    for(int i=0; i<tableRows.length; i++) {
        statement.execute(insertPrefix + tableRows[i]);
    }
    if (close) {
        connection.close();
        return(null);
    } else {
        return(connection);
    }
} catch(SQLException sqle) {
    System.err.println("Error creating table: " + sqle);
    return(null);
}

public static void printTable(String driver,
    String url,
    String username,
    String password,
    String tableName,
    int entryWidth,
    boolean close) {

    String query = "SELECT * FROM " + tableName;
    DBResults results =
        getQueryResults(driver, url, username,
                        password, query, close);
    printTableData(tableName, results, entryWidth, true);
}

/** Prints out all entries in a table. Each entry will
 * be printed in a column that is entryWidth characters
 * wide, so be sure to provide a value at least as big
 * as the widest result.
 */
public static void printTable(Connection connection,
        String tableName,
        int entryWidth,
        boolean close) {
    String query = "SELECT * FROM " + tableName;
    DBResults results =
        getQueryResults(connection, query, close);
    printTableData(tableName, results, entryWidth, true);
}

public static void printTableData(String tableName,
        DBResults results,
        int entryWidth,
        boolean printMetaData) {
    if (results == null) {
        return;
    }
    if (printMetaData) {
        System.out.println("Database: " +
            results.getProductName());
        System.out.println("Version: " +
            results.getProductVersion());
        System.out.println();
    }
    System.out.println(tableName + ":");
    String underline =
        padString("", tableName.length()+1, "=");
    System.out.println(underline);
    int columnCount = results.getColumnCount();
    String separator =
        makeSeparator(entryWidth, columnCount);
    System.out.println(separator);
    String row = makeRow(results.getColumnNames(), entryWidth);
    System.out.println(row);
    System.out.println(separator);
    int rowCount = results.getRowCount();
    for(int i=0; i<rowCount; i++) {
        row = makeRow(results.getRow(i), entryWidth);
        System.out.println(row);
    }
    System.out.println(separator);
}
private static String makeRow(String[] entries, int entryWidth) {
    String row = "|";
    for(int i=0; i<entries.length; i++) {
        row = row + padString(entries[i], entryWidth, " ");
        row = row + " |";
    }
    return(row);
}

private static String makeSeparator(int entryWidth, int columnCount) {
    String entry = padString("", entryWidth+1, "-");
    String separator = "+";
    for(int i=0; i<columnCount; i++) {
        separator = separator + entry + "+";
    }
    return(separator);
}

private static String padString(String orig, int size, String padChar) {
    if (orig == null) {
        orig = "<null>";
    }
    // Use StringBuffer, not just repeated String concatenation
    // to avoid creating too many temporary Strings.
    StringBuffer buffer = new StringBuffer(""");
    int extraChars = size - orig.length();
    for(int i=0; i<extraChars; i++) {
        buffer.append(padChar);
    }
    buffer.append(orig);
    return(buffer.toString());
}
Listing 18.7  DBResults.java

package coreservlets;

import java.sql.*;
import java.util.*;

/** Class to store completed results of a JDBC Query.
 * Differs from a ResultSet in several ways:
 * <UL>
 * <LI>ResultSet doesn’t necessarily have all the data;
 * reconnection to database occurs as you ask for
 * later rows.
 * <LI>This class stores results as strings, in arrays.
 * <LI>This class includes DatabaseMetaData (database product
 * name and version) and ResultSetMetaData
 * (the column names).
 * <LI>This class has a toHTMLTable method that turns
 * the results into a long string corresponding to
 * an HTML table.
 * </UL>
 */

public class DBResults {
    private Connection connection;
    private String productName;
    private String productVersion;
    private int columnCount;
    private String[] columnNames;
    private Vector queryResults;
    String[] rowData;

    public DBResults(Connection connection,
            String productName,
            String productVersion,
            int columnCount,
            String[] columnNames) {
        this.connection = connection;
        this.productName = productName;
        this.productVersion = productVersion;
        this.columnCount = columnCount;
        this.columnNames = columnNames;
        rowData = new String[columnCount];
        queryResults = new Vector();
    }

    public Connection getConnection() {
        return(connection);
    }
}
18.3 Some JDBC Utilities

Listing 18.7 DBResults.java (continued)

```java
public String getProductName() {
    return(productName);
}

public String getProductVersion() {
    return(productVersion);
}

public int getColumnCount() {
    return(columnCount);
}

public String[] getColumnNames() {
    return(columnNames);
}

public int getRowCount() {
    return(queryResults.size());
}

public String[] getRow(int index) {
    return((String[])queryResults.elementAt(index));
}

public void addRow(String[] row) {
    queryResults.addElement(row);
}

/** Output the results as an HTML table, with
 * the column names as headings and the rest of
 * the results filling regular data cells.
 */

public String toHTMLTable(String headingColor) {
    StringBuffer buffer =
    new StringBuffer("<TABLE BORDER=1>
    if (headingColor != null) {
        buffer.append("<TR BGCOLOR=" + headingColor + "
    } else {
        buffer.append("<TR>
    }
    for(int col=0; col<getColumnCount(); col++) {
        buffer.append("<TH>" + columnNames[col]);
```
18.4 Applying the Database Utilities

Now, let’s see how the database utilities of Section 18.3 can simplify the retrieval and display of database results. Listing 18.8 presents a class that connects to the database specified on the command line and prints out all entries in the employees table. Listings 18.9 and 18.10 show the results when connecting to Oracle and Sybase databases, respectively. Listing 18.11 shows a similar class that performs the same database lookup but formats the results in an HTML table. Listing 18.12 shows the raw HTML result. I’ll put an HTML table like this in a real Web page in Section 18.8 (Connection Pooling: A Case Study).

Listing 18.13 shows the JDBC code used to create the employees table.
Listing 18.8 EmployeeTest.java

```java
package coreservlets;

import java.sql.*;

/** Connect to Oracle or Sybase and print "employees" table. */

public class EmployeeTest {
    public static void main(String[] args) {
        if (args.length < 5) {
            printUsage();
            return;
        }
        String vendorName = args[4];
        int vendor = DriverUtilities.getVendor(vendorName);
        if (vendor == DriverUtilities.UNKNOWN) {
            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];
        DatabaseUtilities.printTable(driver, url,
                                       username, password,
                                       "employees", 12, true);
    }

    private static void printUsage() {
        System.out.println("Usage: EmployeeTest host dbName username password oracle|sybase.");
    }
}
```
Chapter 18  JDBC and Database Connection Pooling

Listing 18.9 EmployeeTest result (connecting to Oracle on Solaris)

Prompt> java coreservlets.EmployeeTest dbhost1.apl.jhu.edu PTE
hall xxxx oracle
Database: Oracle
Version: Oracle 7 Server Release 7.2.3.0.0 - Production Release
PL/SQL Release 2.2.3.0.0 - Production

employees:
----------
<table>
<thead>
<tr>
<th>ID</th>
<th>FIRSTNAME</th>
<th>LASTNAME</th>
<th>LANGUAGE</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Wye</td>
<td>Tukay</td>
<td>COBOL</td>
<td>42500</td>
</tr>
<tr>
<td>2</td>
<td>Brit</td>
<td>Tell</td>
<td>C++</td>
<td>62000</td>
</tr>
<tr>
<td>3</td>
<td>Max</td>
<td>Manager</td>
<td>none</td>
<td>15500</td>
</tr>
<tr>
<td>4</td>
<td>Polly</td>
<td>Morphic</td>
<td>Smalltalk</td>
<td>51500</td>
</tr>
<tr>
<td>5</td>
<td>Frank</td>
<td>Function</td>
<td>Common Lisp</td>
<td>51500</td>
</tr>
<tr>
<td>6</td>
<td>Justin</td>
<td>Timecompiler</td>
<td>Java</td>
<td>98000</td>
</tr>
<tr>
<td>7</td>
<td>Sir</td>
<td>Vlet</td>
<td>Java</td>
<td>114750</td>
</tr>
<tr>
<td>8</td>
<td>Jay</td>
<td>Espy</td>
<td>Java</td>
<td>128500</td>
</tr>
</tbody>
</table>

Listing 18.10 EmployeeTest result (connecting to Sybase on NT)

Prompt> java coreservlets.EmployeeTest dbhost2.apl.jhu.edu 605741
hall xxxx sybase
Database: Adaptive Server Anywhere
Version: 6.0.2.2188

employees:
----------
<table>
<thead>
<tr>
<th>id</th>
<th>firstname</th>
<th>lastname</th>
<th>language</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Wye</td>
<td>Tukay</td>
<td>COBOL</td>
<td>42500.0</td>
</tr>
<tr>
<td>2</td>
<td>Brit</td>
<td>Tell</td>
<td>C++</td>
<td>62000.0</td>
</tr>
<tr>
<td>3</td>
<td>Max</td>
<td>Manager</td>
<td>none</td>
<td>15500.0</td>
</tr>
<tr>
<td>4</td>
<td>Polly</td>
<td>Morphic</td>
<td>Smalltalk</td>
<td>51500.0</td>
</tr>
<tr>
<td>5</td>
<td>Frank</td>
<td>Function</td>
<td>Common Lisp</td>
<td>51500</td>
</tr>
<tr>
<td>6</td>
<td>Justin</td>
<td>Timecompiler</td>
<td>Java</td>
<td>98000.0</td>
</tr>
<tr>
<td>7</td>
<td>Sir</td>
<td>Vlet</td>
<td>Java</td>
<td>114750</td>
</tr>
<tr>
<td>8</td>
<td>Jay</td>
<td>Espy</td>
<td>Java</td>
<td>128500.0</td>
</tr>
</tbody>
</table>
Listing 18.11 EmployeeTest2.java

```java
package coreservlets;

import java.sql.*;

/** Connect to Oracle or Sybase and print "employees" table
 * as an HTML table.
 */

class EmployeeTest2 {
    public static void main(String[] args) {
        if (args.length < 5) {
            printUsage();
            return;
        }
        String vendorName = args[4];
        int vendor = DriverUtilities.getVendor(vendorName);
        if (vendor == DriverUtilities.UNKNOWN) {
            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];
        String query = "SELECT * FROM employees";
        DBResults results = DatabaseUtilities.getQueryResults(driver, url,
                    username, password, query, true);
        System.out.println(results.toHTMLTable("CYAN"));
    }

    private static void printUsage() {
        System.out.println("Usage: EmployeeTest2 host dbName username password oracle|sybase.");
    }
}
```

Listing 18.12 EmployeeTest2 result (connecting to Sybase on NT)

Prompt> java coreservlets.EmployeeTest2 dbhost2 605741 hall xxxx sybase

<table>
<thead>
<tr>
<th>id</th>
<th>firstname</th>
<th>lastname</th>
<th>language</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Wye</td>
<td>Tukay</td>
<td>COBOL</td>
<td>42500.0</td>
</tr>
<tr>
<td>2</td>
<td>Britt</td>
<td>Tell</td>
<td>C++</td>
<td>62000.0</td>
</tr>
<tr>
<td>3</td>
<td>Max</td>
<td>Manager</td>
<td>none</td>
<td>15500.0</td>
</tr>
<tr>
<td>4</td>
<td>Polly</td>
<td>Morphic</td>
<td>Smalltalk</td>
<td>51500.0</td>
</tr>
<tr>
<td>5</td>
<td>Frank</td>
<td>Function</td>
<td>Common Lisp</td>
<td>51500.0</td>
</tr>
<tr>
<td>6</td>
<td>Justin</td>
<td>Timecompiler</td>
<td>Java</td>
<td>98000.0</td>
</tr>
<tr>
<td>7</td>
<td>Sir</td>
<td>Vlet</td>
<td>Java</td>
<td>114750.0</td>
</tr>
<tr>
<td>8</td>
<td>Jay</td>
<td>Espy</td>
<td>Java</td>
<td>128500.0</td>
</tr>
</tbody>
</table>

Listing 18.13 EmployeeCreation.java

```java
package coreservlets;
import java.sql.*;
/** Make a simple "employees" table using DatabaseUtilities. */
public class EmployeeCreation {
    public static Connection createEmployees(String driver, String url, String username, String password, boolean close) {
        String format = "(id int, firstname varchar(32), lastname varchar(32), " + "language varchar(16), salary float)");
        String[] employees = {
            "(1, 'Wye', 'Tukay', 'COBOL', 42500)",
            "(2, 'Britt', 'Tell', 'C++', 62000)",
            "(3, 'Max', 'Manager', 'none', 15500)",
            "(4, 'Polly', 'Morphic', 'Smalltalk', 51500)",
            "(5, 'Frank', 'Function', 'Common Lisp', 51500)",
            "(6, 'Justin', 'Timecompiler', 'Java', 98000)",
            "(7, 'Sir', 'Vlet', 'Java', 114750)",
            "(8, 'Jay', 'Espy', 'Java', 128500)",
        }
        return DatabaseUtilities.getConnection(driver, url, username, password, "create table employees (
        id int, firstname varchar(32), lastname varchar(32), language varchar(16), salary float)], close);}
```
18.5 An Interactive Query Viewer

Up to this point, all the database results have been based upon queries that were known at the time the program was written. In many real applications, however, queries are derived from user input that is not known until runtime.
Sometimes the queries follow a fixed format even though certain values change. You should make use of prepared statements in such a case; see Section 18.6 for details. Other times, however, even the query format is variable. Fortunately, this situation presents no problem, since ResultSetMetaData can be used to determine the number, names, and types of columns in a ResultSet, as was discussed in Section 18.1 (Basic Steps in Using JDBC). In fact, the database utilities of Listing 18.6 store that metadata in the DBResults object that is returned from the showQueryData method. Access to this metadata makes it straightforward to implement an interactive graphical query viewer as shown in Figures 18–1 through 18–5. The code to accomplish this result is presented in the following subsection.

![Database Query Viewer](image)

Figure 18–1 Initial appearance of the query viewer.

![Database Query Viewer](image)

Figure 18–2 Query viewer after a request for the complete employees table from an Oracle database.
18.5 An Interactive Query Viewer

Building the display shown in Figures 18–1 through 18–5 is relatively straightforward. In fact, given the database utilities shown earlier, it takes substantially more code to build the user interface than it does to communicate with the database. The full code is shown in Listing 18.14, but I'll give a
quick summary of the process that takes place when the user presses the “Show Results” button.

First, the system reads the host, port, database name, username, password, and driver type from the user interface elements shown. Next, it submits the query and stores the result, as below:

```java
DBResults results =
    DatabaseUtilities.getQueryResults(driver, url,
                                        username, password,
                                        query, true);
```

Next, the system passes these results to a custom table model (see Listing 18.15). If you are not familiar with the Swing GUI library, a table model acts as the glue between a JTable and the actual data.

```java
DBResultsTableModel model = new DBResultsTableModel(results);
JTable table = new JTable(model);
```

Finally, the system places this JTable in the bottom region of the JFrame and calls pack to tell the JFrame to resize itself to fit the table.
package coreservlets;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.table.*;

/** An interactive database query viewer. Connects to
 * the specified Oracle or Sybase database, executes a query,
 * and presents the results in a JTable.
 */

public class QueryViewer extends JFrame
    implements ActionListener{
    public static void main(String[] args) {
        new QueryViewer();
    }

    private JTextField hostField, dbNameField,
        queryField, usernameField;
    private JRadioButton oracleButton, sybaseButton;
    private JPasswordField passwordField;
    private JButton showResultsButton;
    private Container contentPane;
    private JPanel tablePanel;

    public QueryViewer () {
        super("Database Query Viewer");
        WindowUtilities.setNativeLookAndFeel();
        addWindowListener(new ExitListener());
        contentPane = getContentPane();
        contentPane.add(makeControlPanel(), BorderLayout.NORTH);
        pack();
        setVisible(true);
    }

    /** When the "Show Results" button is pressed or
     * RETURN is hit while the query textfield has the
     * keyboard focus, a database lookup is performed,
     * the results are placed in a JTable, and the window
     * is resized to accommodate the table.
     */

    public void actionPerformed(ActionEvent event) {
        String host = hostField.getText();
        String dbName = dbNameField.getText();
        String username = usernameField.getText();
String password =
    String.valueOf(passwordField.getPassword());
String query = queryField.getText();
int vendor;
if (oracleButton.isSelected()) {
    vendor = DriverUtilities.ORACLE;
} else {
    vendor = DriverUtilities.SYBASE;
}
if (tablePanel != null) {
    contentPane.remove(tablePanel);
}
    tablePanel = makeTablePanel(host, dbName, vendor,
        username, password,
        query);
    contentPane.add(tablePanel, BorderLayout.CENTER);
    pack();
}
// Executes a query and places the result in a
// JTable that is, in turn, inside a JPanel.
private JPanel makeTablePanel(String host,
    String dbName,
    int vendor,
    String username,
    String password,
    String query) {
    String driver = DriverUtilities.getDriver(vendor);
    String url = DriverUtilities.makeURL(host, dbName, vendor);
    DBResults results =
        DatabaseUtilities.getQueryResults(driver, url,
            username, password,
            query, true);
    JPanel panel = new JPanel(new BorderLayout());
    if (results == null) {
        panel.add(makeErrorLabel());
        return(panel);
    }
    DBResultsTableModel model =
        new DBResultsTableModel(results);
    JTable table = new JTable(model);
    table.setFont(new Font("Serif", Font.PLAIN, 17));
    table.setRowHeight(28);
    JTableHeader header = table.getTableHeader();
    header.setFont(new Font("SansSerif", Font.BOLD, 13));
    panel.add(table, BorderLayout.CENTER);
    panel.add(header, BorderLayout.NORTH);
18.5 An Interactive Query Viewer

Listing 18.14 QueryViewer.java (continued)

```java
panel.setBorder
    (BorderFactory.createTitledBorder("Query Results"));
return(panel);
}

// The panel that contains the textfields, check boxes,
// and button.
private JPanel makeControlPanel() {
    JPanel panel = new JPanel(new GridLayout(0, 1));
    panel.add(makeHostPanel());
    panel.add(makeUsernamePanel());
    panel.add(makeQueryPanel());
    panel.add(makeButtonPanel());
    panel.setBorder
        (BorderFactory.createTitledBorder("Query Data"));
    return(panel);
}

// The panel that has the host and db name textfield and
// the driver radio buttons. Placed in control panel.
private JPanel makeHostPanel() {
    JPanel panel = new JPanel();
    panel.add(new JLabel("Host:"));
    hostField = new JTextField(15);
    panel.add(hostField);
    panel.add(new JLabel("    DB Name:"));
    dbNameField = new JTextField(15);
    panel.add(dbNameField);
    panel.add(new JLabel("    Driver:"));
    ButtonGroup vendorGroup = new ButtonGroup();
    oracleButton = new JRadioButton("Oracle", true);
    vendorGroup.add(oracleButton);
    panel.add(oracleButton);
    sybaseButton = new JRadioButton("Sybase");
    vendorGroup.add(sybaseButton);
    panel.add(sybaseButton);
    return(panel);
}

// The panel that has the username and password textfields.
// Placed in control panel.
private JPanel makeUsernamePanel() {
    JPanel panel = new JPanel();
    usernameField = new JTextField(10);
    panel.add(usernameField);
```

passwordField = new JPasswordField(10);
panel.add(new JLabel("Username: "));
panel.add(usernameField);
panel.add(new JLabel("Password:");
panel.add(passwordField);
return(panel);
}

// The panel that has textfield for entering queries.
// Placed in control panel.
private JPanel makeQueryPanel() {
    JPanel panel = new JPanel();
    queryField = new JTextField(40);
    queryField.addActionListener(this);
    panel.add(new JLabel("Query:");
    panel.add(queryField);
    return(panel);
}

// The panel that has the "Show Results" button.
// Placed in control panel.
private JPanel makeButtonPanel() {
    JPanel panel = new JPanel();
    showResultsButton = new JButton("Show Results");
    showResultsButton.addActionListener(this);
    panel.add(showResultsButton);
    return(panel);
}

// Shows warning when bad query sent.
private JLabel makeErrorLabel() {
    JLabel label = new JLabel("No Results", JLabel.CENTER);
    label.setFont(new Font("Serif", Font.BOLD, 36));
    return(label);
}
package coreservlets;

import javax.swing.table.*;

/** Simple class that tells a JTable how to extract
 * relevant data from a DBResults object (which is
 * used to store the results from a database query).
 */

class DBResultsTableModel extends AbstractTableModel {
    private DBResults results;

    public DBResultsTableModel(DBResults results) {
        this.results = results;
    }

    public int getRowCount() {
        return(results.getRowCount());
    }

    public int getColumnCount() {
        return(results.getColumnCount());
    }

    public String getColumnName(int column) {
        return(results.getColumnNames()[column]);
    }

    public Object getValueAt(int row, int column) {
        return(results.getRow(row)[column]);
    }
}

Listing 18.15 DBResultsTableModel.java
Listing 18.16 WindowUtilities.java

```java
package coreservlets;

import javax.swing.*;
import java.awt.*;

/** A few utilities that simplify using windows in Swing. */
public class WindowUtilities {

    /** Tell system to use native look and feel, as in previous
     * releases. Metal (Java) LAF is the default otherwise. */
    public static void setNativeLookAndFeel() {
        try {
            UIManager.setLookAndFeel
                (UIManager.getSystemLookAndFeelClassName());
        } catch (Exception e) {
            System.out.println("Error setting native LAF: " + e);
        }
    }

    public static void setJavaLookAndFeel() {
        try {
            UIManager.setLookAndFeel
                (UIManager.getCrossPlatformLookAndFeelClassName());
        } catch (Exception e) {
            System.out.println("Error setting Java LAF: " + e);
        }
    }

    public static void setMotifLookAndFeel() {
        try {
            UIManager.setLookAndFeel
                ("com.sun.java.swing.plaf.motif.MotifLookAndFeel");
        } catch (Exception e) {
            System.out.println("Error setting Motif LAF: " + e);
        }
    }

    }
```
18.6 Prepared Statements (Precompiled Queries)

If you are going to execute similar SQL statements multiple times, using "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 some of 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, and so forth). You then use executeQuery (if you want a ResultSet back) or execute/executeUpdate (for side effects) as with normal statements. For instance, if you were going to give raises to all the personnel in the employees database, you might do something like the following:

```java
Connection connection =
    DriverManager.getConnection(url, user, password);
String template =
    "UPDATE employees SET salary = ? WHERE id = ?";
PreparedStatement statement =
    connection.prepareStatement(template);
float[] newSalaries = getNewSalaries();
```

Chapter 18  JDBC and Database Connection Pooling

int[] employeeIDs = getIDs();
for(int i=0; i<employeeIDs.length; i++) {
    statement.setFloat(1, newSalaries[i]);
    statement.setInt(2, employeeIDs[i]);
    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 18.18 presents a class that sends 40 different queries to a database using prepared statements, then repeats the same 40 queries using regular statements. With a PC and a 28.8K modem connection to the Internet to talk to an Oracle database, prepared statements took only half the time of raw queries, averaging 17.5 seconds for the 40 queries as compared with an average of 35 seconds for the raw queries. Using a fast LAN connection to the same Oracle database, prepared statements took only about 70 percent of the time required by raw queries, averaging 0.22 seconds for the 40 queries as compared with an average of 0.31 seconds for the regular statements. With Sybase, prepared statement times were virtually identical to times for raw queries both with the modem connection and with the fast LAN connection. 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.

Listing 18.18 PreparedStatements.java

package coreservlets;
import java.sql.*;
/** 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.
 */
public class PreparedStatements {
    public static void main(String[] args) {
        if (args.length < 5) {
            printUsage();
            return;
        }
        String vendorName = args[4];
        int vendor = DriverUtilities.getVendor(vendorName);
if (vendor == DriverUtilities.UNKNOWN) {
    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 =
    getConnection(driver, url, username, password);
if (connection != null) {
    doPreparedStatements(connection, print);
    doRawQueries(connection, print);
}

private static void doPreparedStatements(Connection conn,
    boolean print) {
    try {
        String queryFormat =
            "SELECT lastname FROM employees WHERE salary > ?";
        PreparedStatement statement =
            conn.prepareStatement(queryFormat);
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < 40; i++) {
            statement.setFloat(1, i * 5000);
            ResultSet results = statement.executeQuery();
            if (print) {
                showResults(results);
            }
        }
        long stopTime = System.currentTimeMillis();
        double elapsedTime = (stopTime - startTime) / 1000.0;
        System.out.println("Executing prepared statement " +
            "40 times took " + elapsedTime + " seconds.");
    } catch (SQLException sqle) {
        System.out.println("Error executing statement: " + sqle);
    }
}
public static void doRawQueries(Connection conn, boolean print) {
    try {
        String queryFormat =
            "SELECT lastname FROM employees WHERE salary > ";
        Statement statement = conn.createStatement();
        long startTime = System.currentTimeMillis();
        for(int i=0; i<40; i++) {
            String format = queryFormat + (i*5000);
            ResultSet results =
                statement.executeQuery(format);
            if (print) {
                showResults(results);
            }
        }
        long stopTime = System.currentTimeMillis();
        double elapsedTime = (stopTime - startTime)/1000.0;
        System.out.println("Executing raw query 40 times took 
            " + elapsedTime + " seconds.");
    } catch(SQLException sqle) {
        System.out.println("Error executing query: " + sqle);
    }
}

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

private 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);
    }
}
Opening a connection to a database is a time-consuming process. For short queries, it can take much longer to open the connection than to perform the actual database retrieval. Consequently, it makes sense to reuse Connection objects in applications that connect repeatedly to the same database. This section presents a class for connection pooling: preallocating database connections and recycling them as clients connect. Servlets and JSP pages can benefit significantly from this class since the database to which any given servlet or JSP page connects is typically known in advance (e.g., specified in the init method). For example, the servlet shown in Section 18.8 shows a sevenfold performance gain by making use of this connection pool class.

A connection pool class should be able to perform the following tasks:

1. Preallocate the connections.
2. Manage available connections.
3. Allocate new connections.
4. Wait for a connection to become available.
5. Close connections when required.

I'll sketch out the approach to each of these steps here. The full code for the ConnectionPool class is shown in Listing 18.19. As with all classes in the book, you can download the source code from http://www.coreservlets.com/.

1. **Preallocate the connections.**
   Perform this task in the class constructor. Allocating more connections in advance speeds things up if there will be many concurrent requests later but causes an initial delay. As a result, a servlet that preallocates very many connections should build the

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

Listing 18.18 PreparedStatements.java (continued)
connection pool from its `init` method, and you should be sure that the servlet is initialized prior to a “real” client request. The following code uses vectors to store available idle connections and unavailable, busy connections. Assume that `makeNewConnection` uses a URL, username, and password stored previously, then simply calls the `getConnection` method of `DriverManager`.

```java
availableConnections = new Vector(initialConnections);
busyConnections = new Vector();
for (int i=0; i<initialConnections; i++) {
    availableConnections.addElement(makeNewConnection());
}
```

2. **Manage available connections.**

If a connection is required and an idle connection is available, put it in the list of busy connections and then return it. The busy list is used to check limits on the total number of connections as well as when the pool is instructed to explicitly close all connections. One caveat: connections can time out, so before returning the connection, confirm that it is still open. If not, discard the connection and repeat the process. Discarding a connection opens up a slot that can be used by processes that needed a connection when the connection limit had been reached, so use `notifyAll` to tell all waiting threads to wake up and see if they can proceed (e.g., by allocating a new connection).

```java
public synchronized Connection getConnection() throws SQLException {
    if (!availableConnections.isEmpty()) {
        Connection existingConnection =
            (Connection)availableConnections.lastElement();
        int lastIndex = availableConnections.size() - 1;
        availableConnections.removeElementAt(lastIndex);
        if (existingConnection.isClosed()) {
            notifyAll(); // Freed up a spot for anybody waiting.
            return(getConnection()); // Repeat process.
        } else {
            busyConnections.addElement(existingConnection);
            return(existingConnection);
        }
    }
    return(null);
}
```
Allocate new connections.
If a connection is required, there is no idle connection available, and the connection limit has not been reached, then start a background thread to allocate a new connection. Then, wait for the first available connection, whether or not it is the newly allocated one.

```java
if ((totalConnections() < maxConnections) && !connectionPending) { // Pending = connecting in bg
    makeBackgroundConnection();
}
try {
    wait(); // Give up lock and suspend self.
} catch(InterruptedException ie) {}
return(getConnection()); // Try again.
```

Wait for a connection to become available.
This situation occurs when there is no idle connection and you’ve reached the limit on the number of connections. This waiting should be accomplished without continual polling. The natural approach is to use the wait method, which gives up the thread synchronization lock and suspends the thread until notify or notifyAll is called. Since notifyAll could stem from several possible sources, threads that wake up still need to test to see if they can proceed. In this case, the simplest way to accomplish this task is to recursively repeat the process of trying to obtain a connection.

```java
try {
    wait();
} catch(InterruptedException ie) {}
return(getConnection());
```

It may be that you don’t want to let clients wait and would rather throw an exception when no connections are available and the connection limit has been reached. In such a case, do the following instead:

```java
throw new SQLException("Connection limit reached");
```

Close connections when required.
Note that connections are closed when they are garbage collected, so you don’t always have to close them explicitly. But, you sometimes want more explicit control over the process.
public synchronized void closeAllConnections() {
    // The closeConnections method loops down Vector, calling
    // close and ignoring any exceptions thrown.
    closeConnections(availableConnections);
    availableConnections = new Vector();
    closeConnections(busyConnections);
    busyConnections = new Vector();
}

The full class follows.

Listing 18.19 ConnectionPool.java
	nothing
	nothing
	nothing
	nothing
	nothing
	nothing
	nothing
	nothing
	nothing
	nothing

public synchronized Connection getConnection()
    throws SQLException {
    if (!availableConnections.isEmpty()) {
        Connection existingConnection = 
            (Connection)availableConnections.lastElement();
        int lastIndex = availableConnections.size() - 1;
        availableConnections.removeElementAt(lastIndex);
        // If connection on available list is closed (e.g.,
        // it timed out), then remove it from available list
        // and repeat the process of obtaining a connection.
        // Also wake up threads that were waiting for a
        // connection because maxConnection limit was reached.
        if (existingConnection.isClosed()) {
            notifyAll(); // Freed up a spot for anybody waiting
            return(getConnection());
        } else {
            busyConnections.addElement(existingConnection);
            return(existingConnection);
        }
    } else {
        // Three possible cases:
        // 1) You haven’t reached maxConnections limit. So
        // establish one in the background if there isn’t
        // already one pending, then wait for
        // the next available connection (whether or not
        // it was the newly established one).
        // 2) You reached maxConnections limit and waitIfBusy
        // flag is false. Throw SQLException in such a case.
        // 3) You reached maxConnections limit and waitIfBusy
        // flag is true. Then do the same thing as in second
        // part of step 1: wait for next available connection.
        if ((totalConnections() < maxConnections) &&
            !connectionPending) {
            makeBackgroundConnection();
        } else if (!waitIfBusy) {
            throw new SQLException("Connection limit reached");
        }
        // Wait for either a new connection to be established
        // (if you called makeBackgroundConnection) or for
        // an existing connection to be freed up.
        try {
            wait();
            catch(InterruptedException ie) {} // Someone freed up a connection, so try again.
            return(getConnection());
        }
    }
}
// You can't just make a new connection in the foreground
// when none are available, since this can take several
// seconds with a slow network connection. Instead,
// start a thread that establishes a new connection,
// then wait. You get woken up either when the new connection
// is established or if someone finishes with an existing
// connection.

private void makeBackgroundConnection() {
    connectionPending = true;
    try {
        Thread connectThread = new Thread(this);
        connectThread.start();
    } catch(OutOfMemoryError oome) {
        // Give up on new connection
    }
}

public void run() {
    try {
        Connection connection = makeNewConnection();
        synchronized(this) {
            availableConnections.addElement(connection);
            connectionPending = false;
            notifyAll();
        }
    } catch(Exception e) { // SQLException or OutOfMemory
        // Give up on new connection and wait for existing one
        // to free up.
    }
}

// This explicitly makes a new connection. Called in
// the foreground when initializing the ConnectionPool,
// and called in the background when running.

private Connection makeNewConnection() throws SQLException {
    try {
        // Load database driver if not already loaded
        Class.forName(driver);
        // Establish network connection to database
        Connection connection =
            DriverManager.getConnection(url, username, password);
        return(connection);
    } catch(ClassNotFoundException cnfe) {
        // Simplify try/catch blocks of people using this by
        // throwing only one exception type.
        throw new SQLException("Can't find class for driver: " +
            driver);
    }
}
public synchronized void free(Connection connection) {
    busyConnections.removeElement(connection);
    availableConnections.addElement(connection);
    // Wake up threads that are waiting for a connection
    notifyAll();
}

public synchronized int totalConnections() {
    return (availableConnections.size() +
            busyConnections.size());
}

/** Close all the connections. Use with caution:
 *  be sure no connections are in use before
 *  calling. Note that you are not <I>required</I> to
 *  call this when done with a ConnectionPool, since
 *  connections are guaranteed to be closed when
 *  garbage collected. But this method gives more control
 *  regarding when the connections are closed.
 */

public synchronized void closeAllConnections() {
    closeConnections(availableConnections);
    availableConnections = new Vector();
    closeConnections(busyConnections);
    busyConnections = new Vector();
}

private void closeConnections(Vector connections) {
    try {
        for (int i=0; i<connections.size(); i++) {
            Connection connection =
                (Connection)connections.elementAt(i);
            if (!connection.isClosed()) {
                connection.close();
            }
        }
    } catch (SQLException sqle) {
        // Ignore errors; garbage collect anyhow
    }
}

public synchronized String toString() {
    String info =
        "ConnectionPool(" + url + "," + username + ")" +
        ", available=" + availableConnections.size() +
        ", busy=" + busyConnections.size() +
        ", max=" + maxConnections;
    return (info);
}
18.8 Connection Pooling: A Case Study

OK, so we have a ConnectionPool class: what good does it do us? Let’s find out. Listing 18.20 presents a simple servlet that allocates a ConnectionPool in its init method, then, for each request, performs a simple database lookup and places the results in an HTML table. Listing 18.21 and Figure 18–6 show an HTML document that places a copy of this servlet in each of 25 frame cells. Since the servlet stipulates that it not be cached by the browser, this document results in 25 near simultaneous HTTP requests and thus 25 near simultaneous database lookups using connection pooling. This request pattern is similar to what would occur on high-traffic sites even when only a single servlet is used for each page.

Listing 18.22 shows a variation of the servlet that uses a “pool” of only a single connection, and Listing 18.23 shows a third variation that doesn’t use connection pooling at all. Each of these two servlets is also placed in a framed document nearly identical to that of Listing 18.21. Timing results are shown in Table 18.1.

One small reminder: since these servlets load a JDBC driver, the driver needs to be made accessible to the Web server. With most servers, you can make the driver accessible by placing the JAR file containing the driver into the server’s lib directory or by unpacking the JAR file in the classes directory. See your server’s documentation for definitive instructions.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Average Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slow modem connection to database, 10 initial connections, 50 max connections (ConnectionPoolServlet)</td>
<td>11 seconds</td>
</tr>
<tr>
<td>Slow modem connection to database, recycling a single connection (ConnectionPoolServlet2)</td>
<td>22 seconds</td>
</tr>
<tr>
<td>Slow modem connection to database, no connection pooling (ConnectionPoolServlet3)</td>
<td>82 seconds</td>
</tr>
</tbody>
</table>
18.8 Connection Pooling: A Case Study

Table 18.1 Connection pool timing results

<table>
<thead>
<tr>
<th>Condition</th>
<th>Average Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast LAN connection to database, 10 initial connections, 50 max connections (ConnectionPoolServlet)</td>
<td>1.8 seconds</td>
</tr>
<tr>
<td>Fast LAN connection to database, recycling a single connection (ConnectionPoolServlet2)</td>
<td>2.0 seconds</td>
</tr>
<tr>
<td>Fast LAN connection to database, no connection pooling (ConnectionPoolServlet3)</td>
<td>2.8 seconds</td>
</tr>
</tbody>
</table>

Listing 18.20 ConnectionPoolServlet.java

```java
package coreservlets;

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

/**
 * A servlet that reads information from a database and presents it in an HTML table. It uses connection pooling to optimize the database retrieval. A good test case is ConnectionPool.html, which loads many copies of this servlet into different frame cells.
 */

class ConnectionPoolServlet extends HttpServlet {
    private ConnectionPool connectionPool;

    public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        String table;
        try {
            String query = "SELECT firstname, lastname " + " FROM employees WHERE salary > 70000";
            Connection connection = connectionPool.getConnection();
            DBResults results = DatabaseUtilities.getQueryResults(connection, query, false);
            connectionPool.free(connection);
        }
    }
}
```

Listing 18.20 ConnectionPoolServlet.java (continued)

table = results.toHTMLTable("#FFAD00");
} catch(Exception e) {
    table = "Error: " + e;
}
response.setContentType("text/html");
// Prevent the browser from caching the response. See
// Section 7.2 of Core Servlets and JSP for details.
response.setHeader("Pragma", "no-cache"); // HTTP 1.0
response.setHeader("Cache-Control", "no-cache"); // HTTP 1.1
PrintWriter out = response.getWriter();
String title = "Connection Pool Test";
out.println(ServletUtilities.headWithTitle(title) +
    "<BODY BGCOLOR="#FDF5E6">
" +
    "<CENTER>
" +
    table + "\n" +
    "</CENTER>\n</BODY></HTML>");

/** Initialize the connection pool when servlet is
 * initialized. To avoid a delay on first access, load
 * the servlet ahead of time yourself or have the
 * server automatically load it after reboot.
 */

public void init() {
    int vendor = DriverUtilities.SYBASE;
    String driver = DriverUtilities.getDriver(vendor);
    String host = "dbhost2.apl.jhu.edu";
    String dbName = "605741";
    String url = DriverUtilities.makeURL(host, dbName, vendor);
    String username = "hall";
    String password = "xxxx"; // Changed :-)
    try {
        connectionPool =
            new ConnectionPool(driver, url, username, password,
                initialConnections(),
                maxConnections(),
                true);
    } catch(SQLException sqle) {
        System.err.println("Error making pool: " + sqle);
        getServletContext().log("Error making pool: " + sqle);
        connectionPool = null;
    }
}

public void destroy() {
    connectionPool.closeAllConnections();
}
18.8 Connection Pooling: A Case Study

Listing 18.20 ConnectionPoolServlet.java (continued)

```java
/** Override this in subclass to change number of initial
 * connections.
 */
protected int initialConnections() {
    return(10);
}

/** Override this in subclass to change maximum number of
 * connections.
 */
protected int maxConnections() {
    return(50);
}
```

Listing 18.21 ConnectionPool.html

```html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Frameset//EN">
<html>
<head><title>Servlet Connection Pooling: A Test</title></head>
<!-- Causes 25 near simultaneous requests for same servlet. -->
</html>
</head>
<title>Servlet Connection Pooling: A Test</title></head>

<!-- Causes 25 near simultaneous requests for same servlet. -->
</html>
</head>
<title>Servlet Connection Pooling: A Test</title></head>

<frameset rows="*,*,*,*,*" cols="*" frameborder=0 framespacing=0>
<iframe src="/servlet/coreservlets.ConnectionPoolServlet")
<iframe src="/servlet/coreservlets.ConnectionPoolServlet")
<iframe src="/servlet/coreservlets.ConnectionPoolServlet")
<iframe src="/servlet/coreservlets.ConnectionPoolServlet")
<iframe src="/servlet/coreservlets.ConnectionPoolServlet")
</frameset>
<frameset cols="*" frameborder=0 framespacing=0>
<iframe src="/servlet/coreservlets.ConnectionPoolServlet")
<iframe src="/servlet/coreservlets.ConnectionPoolServlet")
<iframe src="/servlet/coreservlets.ConnectionPoolServlet")
<iframe src="/servlet/coreservlets.ConnectionPoolServlet")
<iframe src="/servlet/coreservlets.ConnectionPoolServlet")
</frameset>
<frameset cols="*" frameborder=0 framespacing=0>
<iframe src="/servlet/coreservlets.ConnectionPoolServlet")
<iframe src="/servlet/coreservlets.ConnectionPoolServlet")
<iframe src="/servlet/coreservlets.ConnectionPoolServlet")
<iframe src="/servlet/coreservlets.ConnectionPoolServlet")
<iframe src="/servlet/coreservlets.ConnectionPoolServlet")
</frameset>
<frameset cols="*" frameborder=0 framespacing=0>
<iframe src="/servlet/coreservlets.ConnectionPoolServlet")
<iframe src="/servlet/coreservlets.ConnectionPoolServlet")
<iframe src="/servlet/coreservlets.ConnectionPoolServlet")
<iframe src="/servlet/coreservlets.ConnectionPoolServlet")
<iframe src="/servlet/coreservlets.ConnectionPoolServlet")
</iframe>
```

Figure 18-6 A framed document that forces 25 nearly simultaneous requests for the same servlet.
18.8 Connection Pooling: A Case Study

Listing 18.22 ConnectionPoolServlet2.java

```java
package coreservlets;

/** A variation of ConnectionPoolServlet that uses only
 * a single connection, queueing up all requests to it.
 * Used to compare timing results.
 */
public class ConnectionPoolServlet2
    extends ConnectionPoolServlet {

    protected int initialConnections() {
        return(1);
    }

    protected int maxConnections() {
        return(1);
    }
}
```

Listing 18.23 ConnectionPoolServlet3.java

```java
package coreservlets;

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

/** A variation of ConnectionPoolServlet that does NOT
 * use connection pooling. Used to compare timing
 * benefits of connection pooling.
 */
public class ConnectionPoolServlet3 extends HttpServlet {

    private String url, username, password;

    public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        String table;
        String query =
            "SELECT firstname, lastname " +
            " FROM employees WHERE salary > 70000";
        try {
            Connection connection =
                DriverManager.getConnection(url, username, password);
            Statement stmt = connection.createStatement();
            ResultSet rs = stmt.executeQuery(query);
            ResultSetMetaData md = rs.getMetaData();
            int numColumns = md.getColumnCount();
            table = "id" +
                (numColumns - 1) + "
```

DBResults results =
    DatabaseUtilities.getQueryResults(connection, query, true);

table = results.toHTMLTable("#FFAD00");
} catch(Exception e) {
    table = "Exception: " + e;
}

response.setContentType("text/html");
// Prevent the browser from caching the response. See
// Section 7.2 of Core Servlets and JSP for details.
response.setHeader("Pragma", "no-cache"); // HTTP 1.0
response.setHeader("Cache-Control", "no-cache"); // HTTP 1.1
PrintWriter out = response.getWriter();
String title = "Connection Pool Test (*No* Pooling)";
out.println(ServletUtilities.headWithTitle(title) +
            "<BODY BGCOLOR="#FDF5E6">
            <CENTER>
                table + 
            </CENTER>
        </BODY></HTML>"};

public void init() {
    try {
        int vendor = DriverUtilities.SYBASE;
        String driver = DriverUtilities.getDriver(vendor);
        Class.forName(driver);
        String host = "dbhost2.apl.jhu.edu";
        String dbName = "605741";
        url = DriverUtilities.makeURL(host, dbName, vendor);
        username = "hall";
        password = "xxxx"; // Changed :-)
    } catch(ClassNotFoundException e) {
        System.err.println("Error initializing: " + e);
        getServletContext().log("Error initializing: " + e);
    }
}
18.9 Sharing Connection Pools

In the previous example, each servlet had its own connection pool. This approach makes sense when different servlets perform substantially different tasks and thus talk to different databases. However, it is also quite common for some or all of the servlets on a server to talk to the same database and thus to share a connection pool. There are two main approaches to sharing pools: using the servlet context (a servlet-specific technique) and using static methods or singleton classes (a general Java technique).

Using the Servlet Context to Share Connection Pools

You can call the servlet \texttt{getServletContext} method to get an object of type \texttt{ServletContext} that is shared by all servlets on the server (or within a Web application if your server supports Web applications). This \texttt{ServletContext} object has a \texttt{setAttribute} method that takes a \texttt{String} and an \texttt{Object} and stores the \texttt{Object} in a table with the \texttt{String} as a key. You can obtain the \texttt{Object} at a later time by calling \texttt{getAttribute} with the \texttt{String} (this method returns \texttt{null} if there is no value associated with the key).

So, for example, a group of servlets that all use the books database could share pools by having each servlet perform the following steps:

```java
ServletContext context = getServletContext();
ConnectionPool bookPool =
    (ConnectionPool)context.getAttribute("book-pool");
if (bookPool == null) {
    bookPool = new ConnectionPool(...);
    context.setAttribute("book-pool", bookPool);
}
```
Chapter 18  JDBC and Database Connection Pooling

Using Singleton Classes to Share Connection Pools

Rather than using the ServletContext to share connection pools, you can use normal static methods. For example, you could write a BookPool class with static getPool and setPool methods and have each servlet check BookPool.getPool to see if the value is non-null, instantiating a new ConnectionPool if necessary. However, each servlet has to repeat similar code, and a servlet could accidentally overwrite the shared pool that BookPool.getPool returns.

A better approach is to use a singleton class to encapsulate the desired behavior. A singleton class is simply a class for which only a single instance can be created, enforced through use of a private constructor. The instance is retrieved through a static method that checks if there is already an object allocated, returning it if so and allocating and returning a new one if not. For example, here is the outline of a singleton BookPool class. Each servlet that used it would obtain the connection pool by simply calling BookPool.getInstance().
public class BookPool extends ConnectionPool {
    private static BookPool pool = null;

    private BookPool(...) {
        super(...); // Call parent constructor
        ...
    }

    public static synchronized BookPool getInstance() {
        if (pool == null) {
            pool = new BookPool(...);
        }
        return(pool);
    }
}