Chapter 22. JDBC

Topics in This Chapter

- The seven basic steps in connecting to databases
- Basic database retrieval example
- Some utilities that simplify JDBC usage
- Classes to retrieve and display database results using both plain text and HTML
- An interactive graphical query viewer
- Precompiled queries

JDBC 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. However, 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. 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.

I THINK WE SHOULD BUILD AN SQL DATABASE.
UH-OH
DOES HE UNDERSTAND WHAT HE SAID OR IS IT SOMETHING HE SAW IN A TRADE MAGAZINE AD?
WHAT COLOR DO YOU WANT THAT DATABASE?
I THINK MAUVE HAS THE MOST RAM.

DILBERT © UFS. Reprinted with permission.

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'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 any purpose on non-Microsoft operating systems as well as free for educational or research use on Windows. For details, see http://www.mysql.com/.

22.1 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 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 representing 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:

```
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://industry.java.sun.com/products/jdbc/drivers. 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. For example, most recent servlet and JSP engines automatically add JAR files that are in the `lib` directory to their `CLASSPATH`. So, if your server-side programs are using JDBC, the JAR files containing the drivers should go in the server's `lib` directory.

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:

```
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 + ":" + "?SERVICEid=" + 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, for JDBC to be used 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. We are 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);
```

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 that 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:

```
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 22.6), `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:

```
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:

```
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 22.6 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 data 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`.

```
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), `getMetaData` (retrieve information about the `ResultSet` in a `ResultSetMetaData` object), and `wasNull` (was the last `getXxx` result SQL `NULL`? Alternatively, for strings you can simply compare the return value to `null`).

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), `getColumnName` (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 other methods 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:

```
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.

22.2 Basic JDBC Example

Listing 22.3 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 22.1 and 22.2. 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 22.4. 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 22.5. For now, just skim quickly over this listing—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 `cwp` package, it resides in a subdirectory called `cwp`. Before compiling the file, we set our `CLASSPATH` to include the directory *containing* the `cwp` directory (the JAR file containing the JDBC drivers should be in the `CLASSPATH` also, of course). With this setup, we compile simply by doing "`javac FruitTest.java`" from within the `cwp` subdirectory. But to run `FruitTest`, we need to refer to the full package name with "`java` **`cwp.`**`FruitTest` ...".

Listing 22.1 `FruitTest` result (connecting to Oracle on Solaris)

```
Prompt> java cwp.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
```

Listing 22.2 `FruitTest` result (connecting to Sybase on NT)

```
Prompt> java cwp.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 22.3 `FruitTest.java`

```
package cwp;

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;
    }
    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()) {
        // 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 22.4 `DriverUtilities.java`

```
package cwp;

/** 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;

  /** Build a URL in the format needed by the
   *  Oracle and Sybase drivers we are 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" +
             "?SERVICEid=" + 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 22.5 **`FruitCreation.java`**

```
package cwp;

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.");
  }
}
```

22.3 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, in an Excel spreadsheet, 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 22.7). 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 22.6 gives the main code, and Listing 22.7 presents the auxiliary DBResults class that stores the accumulated results and return 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.

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

Since an HTML table can do double duty as an Excel spreadsheet, 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.corewebprogramming.com and used or adapted without restriction.

Listing 22.6 DatabaseUtilities.java

```
package cwp;

import java.sql.*;

public class DatabaseUtilities {

  /** Connect to database, execute specified query,
   *  and accumulate results into DBRresults 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) {
    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();
          }
          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);
  }

  // A String of the form "|  xxx |  xxx |  xxx |"

  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);
  }

  // A String of the form "+------+------+------+"

  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 22.7 `DBResults.java`

```
package cwp;

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);
  }

  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>\n");
    if (headingColor != null) {
      buffer.append("  <TR BGCOLOR=\"" + headingColor +
                    "\">\n    ");
    } else {
      buffer.append("  <TR>\n    ");
    }
    for(int col=0; col<getColumnCount(); col++) {
      buffer.append("<TH>" + columnNames[col]);
    }
    for(int row=0; row<getRowCount(); row++) {
      buffer.append("\n  <TR>\n    ");
      String[] rowData = getRow(row);
      for(int col=0; col<getColumnCount(); col++) {
        buffer.append("<TD>" + rowData[col]);
      }
    }
    buffer.append("\n</TABLE>");
    return(buffer.toString());
  }
}
```

22.4 Applying the Database Utilities

Now, let's see how the database utilities of Section 22.3 can simplify the retrieval and display of database results. Listing 22.8 presents a class that connects to the database specified on the command line and prints out all entries in the `employees` table. Listings 22.9 and 22.10 show the results when connecting to Oracle and Sybase databases, respectively. Listing 22.11 shows a similar class that performs the same database lookup but formats the results in an HTML table. Listing 22.12 shows the raw HTML result.

Listing 22.13 shows the JDBC code used to create the `employees` table.

Listing 22.8 `EmployeeTest.java`

```
package cwp;
```

```
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.");
  }
}
```

Listing 22.9 `EmployeeTest` result (connecting to Oracle on Solaris)

```
Prompt> java cwp.EmployeeTest 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

employees:
==========
+-------------+-------------+-------------+-------------+-------------+
|          ID |   FIRSTNAME |    LASTNAME |    LANGUAGE |      SALARY |
+-------------+-------------+-------------+-------------+-------------+
|           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 |
+-------------+-------------+-------------+-------------+-------------+
```

Listing 22.10 `EmployeeTest` **result (connecting to Sybase on NT)**

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

employees:
==========
+-------------+-------------+-------------+-------------+-------------+
|          id |   firstname |    lastname |    language |      salary |
+-------------+-------------+-------------+-------------+-------------+
|           1 |         Wye |       Tukay |       COBOL |     42500.0 |
|           2 |       Britt |        Tell |         C++ |     62000.0 |
|           3 |         Max |     Manager |        none |     15500.0 |
|           4 |       Polly |     Morphic |   Smalltalk |     51500.0 |
|           5 |       Frank |    Function | Common Lisp |     51500.0 |
|           6 |      Justin |Timecompiler |        Java |     98000.0 |
|           7 |         Sir |        Vlet |        Java |    114750.0 |
|           8 |         Jay |        Espy |        Java |    128500.0 |
+-------------+-------------+-------------+-------------+-------------+
```

Listing 22.11 `EmployeeTest2.java`

```
package cwp;

import java.sql.*;

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

public 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 22.12 `EmployeeTest2` result (connecting to Sybase on NT)

```
Prompt> java cwp.EmployeeTest2 dbhost2 605741
        hall xxxx sybase
<TABLE BORDER=1>
  <TR BGCOLOR="CYAN">
    <TH>id<TH>firstname<TH>lastname<TH>language<TH>salary
  <TR>
    <TD>1<TD>Wye<TD>Tukay<TD>COBOL<TD>42500.0
  <TR>
    <TD>2<TD>Britt<TD>Tell<TD>C++<TD>62000.0
  <TR>
    <TD>3<TD>Max<TD>Manager<TD>none<TD>15500.0
  <TR>
    <TD>4<TD>Polly<TD>Morphic<TD>Smalltalk<TD>51500.0
  <TR>
    <TD>5<TD>Frank<TD>Function<TD>Common Lisp<TD>51500.0
  <TR>
    <TD>6<TD>Justin<TD>Timecompiler<TD>Java<TD>98000.0
  <TR>
    <TD>7<TD>Sir<TD>Vlet<TD>Java<TD>114750.0
  <TR>
    <TD>8<TD>Jay<TD>Espy<TD>Java<TD>128500.0
</TABLE>
```

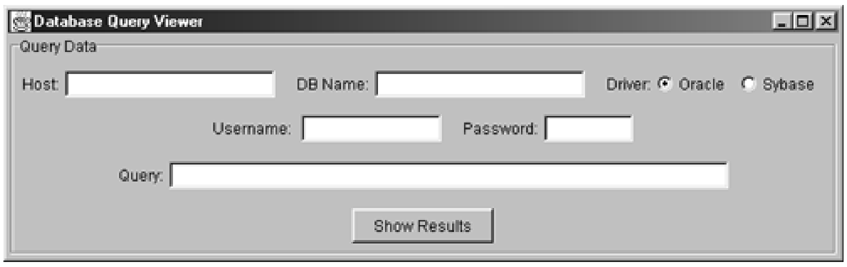
Listing 22.13 `EmployeeCreation.java`

```
package cwp;

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.createTable(driver, url,
                                         username, password,
                                         "employees",
                                         format, employees,
                                         close));
  }

  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];
    createEmployees(driver, url, username, password, true);
  }

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

22.5 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 run time. 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 22.6 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 22.1 (Basic Steps in Using JDBC). In fact, the database utilities of Listing 22.6 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 22-1 through 22-5. The code to accomplish this result is presented in the following subsection.

Figure 22-1. Initial appearance of the query viewer.



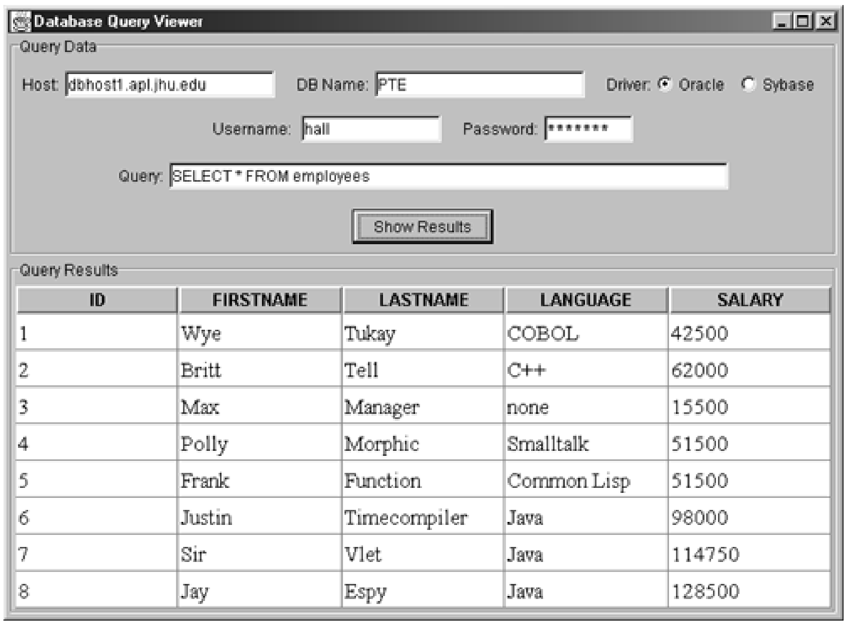
Query Data

Host: | DB Name: | Driver: [✓ Oracle] Sybase

Username: | Password:

Query:

Figure 22-2. Query viewer after a request for the complete `employees` table from an Oracle database.



Query Data

Host: dbhost1.apl.jhu.edu | DB Name: PTE | Driver: [✓ Oracle] Sybase

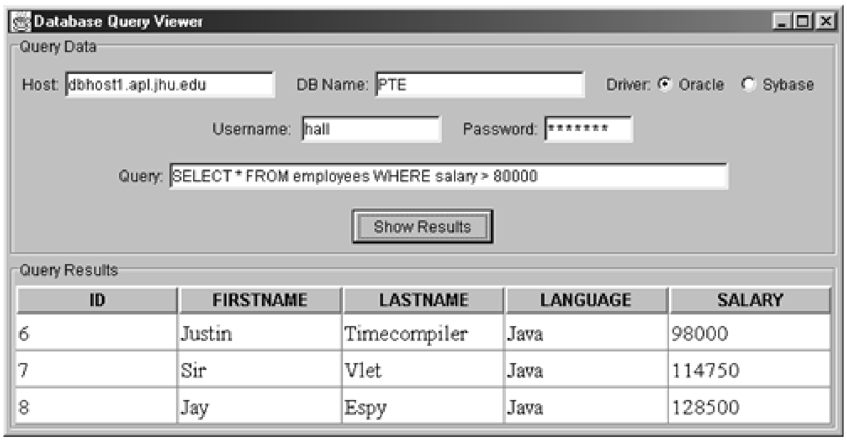
Username: hall | Password: *******

Query: SELECT * FROM employees

Query Results

ID	FIRSTNAME	LASTNAME	LANGUAGE	SALARY
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

Figure 22-3. Query viewer after a request for part of the `employees` table from an Oracle database.



Query Data

Host: dbhost1.apl.jhu.edu | DB Name: PTE | Driver: [✓ Oracle] Sybase

Username: hall | Password: *******

Query: SELECT * FROM employees WHERE salary > 80000

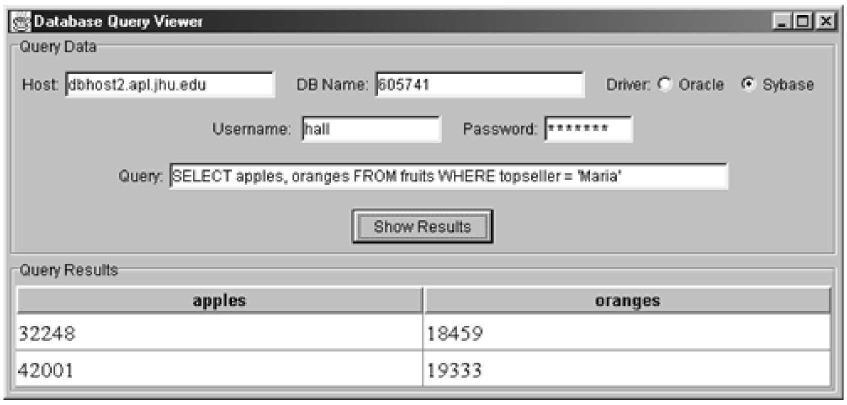
Query Results

ID	FIRSTNAME	LASTNAME	LANGUAGE	SALARY
6	Justin	Timecompiler	Java	98000
7	Sir	Vlet	Java	114750
8	Jay	Espy	Java	128500

Figure 22-4. Query viewer after a request for the complete `fruits` table from a Sybase database.

graphics/22fig04.gif

Figure 22-5. `fruits` table from a Sybase database.


Database Query Viewer

Query Data

Host: dbhost2.apl.jhu.edu DB Name: 605741 Driver: Oracle [✓ Sybase]

Username: hall Password: *******

Query: SELECT apples, oranges FROM fruits WHERE topseller = 'Maria'

Query Results

apples	oranges
32248	18459
42001	19333

Query Viewer Code

Building the display shown in Figures 22-1 through 22-5 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 22.14, but we'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:

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

Next, the system passes these results to a custom table model (see Listing 22.15). If you are not familiar with the Swing GUI library, note that a table model acts as the glue between a `JTable` and the actual data.

```
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.

Listing 22.14 `QueryViewer.java`

```
package cwp;

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);
    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);
    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);
  }
}
```

Listing 22.15 `DBResultsTableModel.java`

```
package cwp;

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).
 */

public 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]);
  }
}
```

22.6 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 `set`*`Xxx`* 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` (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:

```
Connection connection =
  DriverManager.getConnection(url, user, password);
String template =
  "UPDATE employees SET salary = ? WHERE id = ?";
PreparedStatement statement =
  connection.prepareStatement(template);
float[] newSalaries = getNewSalaries();
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 22.16 presents a class that sends 40 different queries to a database by means of prepared statements, then repeats the same 40 queries with 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. When a fast LAN connection to the same Oracle database was used, 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 the Sybase driver we used, 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.corewebprogramming.com/, add information about your drivers to it, then run the `PreparedStatements` program yourself.

Listing 22.16 `PreparedStatements.java`

```
package cwp;

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++) {
        ResultSet results =
          statement.executeQuery(queryFormat + (i*5000));
        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);
    }
  }

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

22.7 Summary

JDBC provides a standard way of accessing relational databases from programs written in the Java programming language. It lets you avoid vendor-specific code, thus simplifying the process of using multiple databases and switching from one database vendor to another.

Although JDBC standardizes the mechanism for connecting to the database and the data structure that represents the result, it does not standardize SQL syntax. This means that you can still use vendor-specific SQL commands if you want to; it also means that SQL queries and commands need to be built from raw strings rather than by means of SQL-related method calls.

JDBC can be used from desktop applications and applets, although with applets you are restricted to the case where the database server runs on the same host as the Web server. However, JDBC is most commonly used from server-side applications such as servlets and JSP. In fact, in many server-side applications, most of the real work is done in the database; servlets and JSP just provide a convenient middle tier for passing the data from the browser to the database and formatting the results that come back from the database.

CONTENTS