

Chapter 3: JDBC—The Java Database API

In this chapter

- What Is JDBC?
- JDBC Core Components
- A Simple Database Query Program
- Inserting, Updating, and Deleting Data
- Updating Data from a Result Set
- The JDBC Optional Package
- Troubleshooting

What Is JDBC?

The *Java Database Connectivity* (JDBC) API is one of the most important APIs for enterprise-level development because you almost always need to access a database. JDBC gives you a standard API that is mostly database-independent, but still allows you to access specific features of your database if necessary.

There are actually two parts to the JDBC API. The core JDBC API (`java.sql.*`) comes with the standard Java Development Kit. J2EE includes the JDBC Optional Package (`javax.sql.*`) that includes some features more commonly used for J2EE development (especially in the area of Enterprise JavaBeans).

Most databases have very different APIs for communicating with the database. On the Windows platform and even some Unix platforms, the ODBC API (Open Database Connectivity) gives you a standard database API that works with many different databases. JDBC solves the same problem as ODBC because it also gives you a standard database API.

Like ODBC, the JDBC package itself doesn't know how to connect to any database. It is an API framework that relies on other packages to provide the implementation. You can go to <http://www.oracle.com> or <http://www.informix.com> and download JDBC drivers that work with the Oracle and Informix databases. No matter what database you use, there's a good chance that there's already a JDBC driver available for it.

There are four types of JDBC drivers, called Type 1, Type 2, Type 3, and Type 4. It's important to know the various types when you first choose a driver, and sometimes the driver choice might affect your application design—especially if you are developing Java applets.

The first distinction to draw among the four types is that Type 1 and Type 2 drivers involve native libraries—they aren't pure Java. This means it might be more difficult to find a driver for your hardware platform, and it also means you typically can't use the driver from a Java applet. Technically, you can't use the driver from an *unsigned* applet, but a *signed* applet might be able to use the driver. For more

about signing applets, see Chapter 45, "Code Signing."

Type 1 JDBC Drivers

A Type 1 JDBC driver uses a native library with a common interface. That is, the native library isn't database specific. The most common example of a Type 1 driver is the JDBC-ODBC bridge that comes with the JDK. The bridge doesn't need to know about every kind of database; it only needs to know how to use the ODBC API.

[Figure 3.1](#) illustrates a typical Type 1 driver configuration.

Figure 3.1

A Type 1 driver uses a native library to communicate with a database-independent API.

Although they use native code, Type 1 drivers still tend to be slow because the data must pass through so many layers. ODBC, for example, still needs a database-specific driver, so your data passes through the database-specific driver, the ODBC driver, and finally the JDBC driver before it reaches you.

Type 2 JDBC Drivers

A Type 2 driver accesses a database-specific driver through a native library. Because it uses a native library, a Type 2 driver is often fairly quick, although there is still some slowdown in the interface between Java and the native API. As with the Type 1 driver, the native library tends to limit your cross-platform options because you might not be able to find a driver for your hardware platform.

[Figure 3.2](#) illustrates a typical Type 2 driver configuration.

Figure 3.2

A Type 2 driver uses a database-specific native library.

Type 3 JDBC Drivers

A Type 3 JDBC driver is pure Java and uses a database-independent protocol to communicate with a database gateway. You typically use a Type 3 driver and database gateway when you develop Java applets because the gateway helps you work around some of the applet security restrictions. [Figure 3.3](#) illustrates a typical Type 3 driver configuration.

Figure 3.3

A Type 3 driver communicates with a database gateway.

Using a Type 3 driver can be one of the slowest ways to access data because of the presence of the database gateway. The gateway must read the data from the database and then send it to you. It doubles the amount of network traffic, and networking tends to be one of the slower parts of an application.

Type 4 JDBC Drivers

A Type 4 driver is pure Java and communicates directly with the database. In the very early days of Java, before Just-In-Time (JIT) compilers were available, Type 2 drivers were the most popular drivers because of their speed. Type 4 drivers are now the most popular because the JIT makes the driver

perform at levels comparable to the native driver, and because the data doesn't pass through the JNI layer (that is, the driver doesn't need to translate data into Java objects), the Type 4 drivers typically outperform Type 2 drivers. Plus, the Type 4 drivers work on any Java platform. Of course, the Type 4 drivers are database-specific, so you need a different driver for each different database platform. A Type 4 driver for Oracle can't access an Informix database. [Figure 3.4](#) illustrates a typical Type 4 driver configuration.

Keep in mind that your choice of driver doesn't change how you write your code. A large number of applications allow you to specify the JDBC driver at runtime. The API itself doesn't care about the type of driver.

Figure 3.4

A Type 4 driver communicates directly with the database.

JDBC Core Components

[Figure 3.5](#) illustrates the major classes of the JDBC API and how they relate to each other.

Figure 3.5

The major components of the JDBC API are shown here.

Of the major JDBC components, only `DriverManager` is a concrete Java class. The rest of the components are Java interfaces that are implemented by the various driver packages.

DriverManager

The `DriverManager` class keeps track of the available JDBC drivers and creates database connections for you. Although the JDBC driver itself creates the database connection, you usually go through the `DriverManager` to get the connection. That way you never need to deal with the actual driver class.

The `DriverManager` class has a number of useful static methods, the most popular of which is `getConnection`:

```
public static Connection getConnection(String url)
public static Connection getConnection(String url,
    String username, String password)
public static Connection getConnection(String url,
    Properties props)
```

The `url` parameter in the `getConnection` method is one of the key features of JDBC. It specifies what database you want to use. The general form of the JDBC URL is

```
jdbc:drivertype:driversubtype://params
```

The *:driversubtype* part of the URL is optional. You might just have a URL of the form

```
jdbc:drivertype://params
```

For an ODBC database connection, the URL takes the form

```
jdbc:odbc:datasourcename
```

Consult the documentation for your JDBC driver to see the exact format of the driver's URL. The only thing you can count on is that it will start with `jdbc:`.

The `DriverManager` class knows about all the available JDBC drivers—at least the ones available in your program. There are two ways to tell the `DriverManager` about a JDBC driver. The first is to load the driver class. Most JDBC drivers automatically register themselves with the `DriverManager` as soon as their class is loaded (they do the registration using a static initializer). For example, to register the JDBC-ODBC driver that comes with the JDK, you can use the statement:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Although this method seems a little quirky, it's extremely common. Keep in mind that if the driver isn't in your classpath, the `Class.forName` method throws a `ClassNotFoundException`.

The other way to specify the available drivers is by setting the `jdbc.drivers` system property. This property is a list of driver class names separated by colons. For example, when you run your program, you might include the property

```
java -Djdbc.drivers=sun.jdbc.odbc.JdbcOdbcDriver:  
COM.cloudscape.core.JDBCdriver MyJDBCProgram
```

The `DriverManager` class also performs some other useful functions. Many database programs need to log data—especially database statements. Although you don't often log database statements in a production system, you frequently need to in a development environment. The `setLogWriter` method in the `DriverManager` class lets you specify a `PrintWriter` that will be used to log any JDBC-related information. The `DriverManager` class also supplies a `println` method for writing to the database log and a `getLogWriter` method to give you direct access to the log writer object.

Driver

The `Driver` interface is primarily responsible for creating database connections. The `connect` method returns a `Connection` object representing a database connection:

```
public Connection connect(String url, Properties props)
```

Connection

The `Connection` interface represents the core of the JDBC API. You can group most of the methods in the `Connection` interface into three major categories:

- Getting database information
- Creating database statements
- Managing database transactions

Getting Database Information

The `getMetaData` method in the `Connection` interface returns a `DatabaseMetaData` object that describes the database. You can get a list of all the database tables and examine the definition of each table. You probably won't find the metadata too useful in a typical database application, but it's great for writing database explorer tools that gather information about the database.

Creating Database Statements

You use statements to execute database commands. There are three types of statements: `Statement`, `PreparedStatement`, and `CallableStatement`. Each of these statements works in a slightly different way, but the end result is always that you execute a database command.

The methods for creating a `Statement` are

```
public Statement createStatement()  
public Statement createStatement(int resultSetType,  
    int resultSetConcurrency)
```

When you create any kind of statement, you can specify a particular result set type and concurrency. These values determine how the connection handles the results returned by a query. Specifically, the `resultSetType` parameter lets you create a scrollable result set so you can move forward and backward through the results. By default, you can only move forward through the results. The `resultSetConcurrency` parameter lets you specify whether you can update the result set. By default, the result set is read-only.

The methods for creating a `PreparedStatement` are

```
public PreparedStatement prepareStatement(String sql)  
public PreparedStatement prepareStatement(String sql,  
    int resultSetType, int resultSetConcurrency)
```

The methods for creating a `CallableStatement` are

```
public CallableStatement prepareCall(String sql)  
public CallableStatement prepareCall(String sql,  
    int resultSetType, int resultSetConcurrency)
```

Managing Transactions

Normally, every statement you execute is a separate database transaction. Sometimes, however, you want to group several statements into a single transaction. The `Connection` class has an auto-commit flag that indicates whether it should automatically commit transactions or not. If you want to define your own transaction boundaries, call

```
public void setAutoCommit(boolean autoCommitFlag)
```

After you turn off auto-commit, you can start executing your statements. When you reach the end of your transaction, call the `commit` method to commit the transaction (complete it):

```
public void commit()
```

If you decide that you don't want to complete the transaction, call the `rollback` method to undo the

changes made in the current transaction:

```
public void rollback()
```

When you are done with a connection, make sure you close it by calling the `close` method:

```
public void close()
```

Statement

As you now know, there are three different kinds of JDBC statements: `Statement`, `PreparedStatement`, and `CallableStatement`.

The `Statement` interface defines methods that allow you to execute an SQL statement contained within a string. The `executeQuery` method executes an SQL string and returns a `ResultSet` object, while the `executeUpdate` executes an SQL string and returns the number of rows updated by the statement:

```
ResultSet executeQuery(String sqlQuery)
```

You usually use `executeQuery` when you execute an SQL `SELECT` statement, and you use `executeUpdate` when you execute an SQL `UPDATE`, `INSERT`, or `DELETE` statement:

```
int executeUpdate(String sqlUpdate)
```

When you are done with a statement, make sure you close it. If not, you might soon run out of available statements. A database connection usually allows a specific number of open statements. If the garbage collector hasn't destroyed the old connections you aren't using, you might exceed the maximum number of statements. To close a statement, just call the statement's `close` method:

```
public void close()
```

Note

When you close a `Connection`, it automatically closes any open statements or result sets.

Although the `Statement` interface contains a lot of methods for accessing results and setting various parameters, you will likely find that you only use the two `execute` methods and the `close` method in most of your applications.

Although the `Statement` interface is the simplest of the three statement interfaces, it can often cause you some programming headaches. Most of the time, you aren't executing exactly the same statement—you build a statement based on the data you are looking for or changing. In other words, you don't search for all people with a last name of Smith every time you do a query. You just search for all people with some specific last name.

If you just use the `Statement` interface, your query often looks something like this:

```
ResultSet results = stmt.executeQuery(
    "select * from Person where last_name = '"+
    lastNameParam+"'");
```

This code looks a little ugly because you've got the single quotes (for the SQL string) inside the double quotes for the Java string. Now, what happens if `lastNameParam` is O'Brien? Your SQL string would be

```
select * from Person where last_name = 'O'Brien'
```

The database would give you an error because you really need two single quotes in O'Brien (that is, O'Brien). You sometimes end up writing an `escapeQuotes` routine that looks for single quotes in a string and replaces them with two single quotes. Your `executeQuery` call would then look like this:

```
ResultSet results = stmt.executeQuery(escapeQuotes(
    "select * from Person where last_name = '"+
    lastNameParam+"'"));
```

You can handle this much better by using a prepared statement. A prepared statement is an SQL statement with parameters you can change at any time. For example, you would create a prepared statement with your last name search using the following call:

```
PreparedStatement pstmt = myConnection.prepareStatement(
    "select * from Person where last_name = ?");
```

Now, when you go to perform the query, you use one of the `set` methods in the `PreparedStatement` interface to store the value for the parameter (the `?` in the query string). You can have more than one parameter in a prepared statement. Now, to query for people whose last name is O'Brien, you use the following calls:

```
pstmt.setString(1, "O'Brien");
ResultSet results = pstmt.executeQuery();
```

The `PreparedStatement` interface has `set` methods for most Java data types and also allows you to store `BLOBS` and `CLOBs` using various data streams. The first parameter for each of the `set` methods is the parameter number. The first parameter is always 1 (not 0, as is the case with arrays and other data sequences). Also, to store a `NULL` value, you must indicate the data type of the column you are setting to `NULL`. For example, to set an integer value to `NULL`, use this call:

```
pstmt.setNull(1, Types.INTEGER);
```

The `Types` class contains constants that represent the various SQL data types supported by JDBC.

You can reuse a prepared statement. That is, after you have executed it, you can execute it again, or you can first change some of the parameter values and then execute it. Some applications create their prepared statements ahead of time, although many still create them when needed.

Usually, you can create the statements ahead of time only if you are using a single database connection, because statements are always associated with a specific connection. If you have a pool of database connections, you need to create the prepared statement when you actually need to use it, because you might get a different database connection from the pool each time.

The `CallableStatement` interface is used to access SQL stored procedures (SQL code stored on the database). The `CallableStatement` interface lets you invoke stored procedures and retrieve any results returned by the stored procedure. Stored procedures, incidentally, let you write queries that can run very quickly and are easy to invoke. It is often easier to update your application by changing a few stored procedures. The disadvantage, however, is that every database has a different syntax for stored procedures, so if you need to migrate from one database to another, you must rewrite all your stored procedures.

Tip

The Oracle database lets you write stored procedures in Java! You don't need to learn a new language to write Oracle stored procedures.

JDBC has a standard syntax for executing stored procedures, which takes one of two forms:

```
{call procedurename param1, param2, param3 ... }
{?= call procedurename param1, param2, param3 ... }
```

The parameters are optional. If your procedure doesn't take any parameters, the call might look like this:

```
{call myprocedure}
```

If your stored procedure returns a value, use the form that starts with `?=`. You can also use `?` for any of the parameter values in the stored procedure call and set them just like you set parameters in a `PreparedStatement`. In fact, the `CallableStatement` interface extends the `PreparedStatement` interface.

Some stored procedures have a notion of "out parameters," in which you pass a parameter in, the procedure changes the value of the parameter, and you need to examine the new value. If you need to retrieve the value of a parameter, you must tell the `CallableStatement` interface ahead of time by calling `registerOutParameter`:

```
public void registerOutParameter(int whichParameter, int sqlType)
public void registerOutParameter(int whichParameter, int sqlType,
    int scale)
public void registerOutParameter(int whichParameter, int sqlType,
    String typeName)
```

After you execute your stored procedure, you can retrieve the values of the out parameters by calling one of the many `get` methods. As with the `set` methods, the parameter numbers on the `get` methods are numbered starting from 1 and not 0. For example, suppose you have a stored procedure called `findPopularName` that searches the `Person` table for the most popular first name. Suppose, furthermore, that the stored procedure has a single out parameter that is the most popular name. You invoke the procedure this way:

```
CallableStatement cstmt = myConnection.prepareCall(
    "{call findPopularName ?}");
cstmt.registerOutParameter(1, Types.VARCHAR);
```

```
cstmt.execute();
System.out.println("The most popular name is "+
    cstmt.getString(1));
```

ResultSet

The `ResultSet` interface lets you access data returned from an SQL query. The most common use of the `ResultSet` interface is just to read data, although you can also update rows and delete rows as of JDBC version 2.0. If you just want to read results, use the `next` method to move to the next row and then use any of the numerous `get` methods to retrieve the data. For example

```
ResultSet results = stmt.executeQuery(
    "select last_name, age from Person");
while (results.next())
{
    String lastName = results.getString("last_name");
    int age = results.getInt("age");
}
```

Each of the `get` methods in the `ResultSet` interface lets you specify which item you want by the column name or by the position in the query. For example, when you query for `last_name`, `age`, the `last_name` column is in the first position and `age` is in the second position. You can retrieve the `last_name` value with

```
String lastName = results.getString(1);
```

Retrieving a result by index is faster than retrieving a result by column name. When you retrieve a result by column name, the result set must first determine the index that corresponds to the column name. The disadvantage of using the index is that your code is a little harder to maintain. If you change the order of the columns or insert new columns, you must remember to update the indexes. If you use column names, you don't need to change existing code if you add new columns to the query or change the order of the columns. Use the column names when possible, but switch to using an index when you need to improve performance.

Tip

You can use the `findColumn` method in the result set to determine the index for a particular column name. If you have a query that returns a large number of rows, you should consider using `findColumn` to determine the indexes first, then retrieve the values using the index instead of the column name. Your program will be faster, but you still will have the benefits of using column names.

A Simple Database Query Program

Listing 3.1 shows a simple JDBC query program. The database used for this example is the Cloudscape database, a 100% pure Java database available from <http://www.cloudscape.com>. The program puts together the various concepts you have already seen in this chapter.

Listing 3.1 Source Code for `SimplyQuery.java`

```
package usingj2ee.jdbc;

import java.sql.*;

public class SimpleQuery
{
    public static void main(String[] args)
    {
        try
        {
            // Make sure the DriverManager knows about the driver
            Class.forName("COM.cloudscape.core.JDBCdriver");

            // Create a connection to the database
            Connection conn = DriverManager.getConnection(
                "jdbc:cloudscape:j2eebook");

            // Create a statement
            Statement stmt = conn.createStatement();

            // Execute the query
            ResultSet results = stmt.executeQuery(
                "select * from person");

            // Loop through all the results
            while (results.next())
            {
                // Get the values from the result set
                String firstName = results.getString("first_name");
                String middleName = results.getString("middle_name");
                String lastName = results.getString("last_name");
                int age = results.getInt("age");

                // Print out the values
                System.out.println(firstName+" "+middleName+" "+lastName+
                    " "+age);
            }

            conn.close();
        }
        catch (Exception exc)
        {
            exc.printStackTrace();
        }
    }
}
```

Tip

Don't forget to include the JDBC driver in your classpath before running the example. If you are using a database other than Cloudscape, you must also change both the driver name and the database URL.

Inserting, Updating, and Deleting Data

After you know the SQL commands, it's not hard to make database updates. The pattern for performing inserts, updates, and deletions is basically the same. You can either use the `Statement` or the `PreparedStatement` interface, depending on whether you want to insert the data into the SQL string or use parameterized data.

Listing 3.2 shows a program that inserts, updates, and then deletes a row. The example uses the original definition of the `Person` table from Chapter 2, "A Quick Primer on SQL."

Listing 3.2 Source Code for `InsUpdDel.java`

```
package usingj2ee.jdbc;

import java.sql.*;

public class InsUpdDel
{
    public static void main(String[] args)
    {
        try
        {
            // Make sure the DriverManager knows about the driver
            Class.forName("COM.cloudscape.core.JDBCdriver");

            // Create a connection to the database
            Connection conn = DriverManager.getConnection(
                "jdbc:cloudscape:j2eebook");

            // Create a prepared statement for inserting data
            // In case you are wondering about the efficiency of concatenating
            // strings at runtime, if you just have a series of constant strings with
            // no variables in between, the compiler automatically combines the
            // strings
            PreparedStatement pstmt = conn.prepareStatement(
                "insert into SSN_Info (first_name, middle_name, last_name, "+
                "ssn) values (?, ?, ?, ?)");

            // Store the column values for the new table row
            pstmt.setString(1, "argle");
            pstmt.setString(2, "quinton");
            pstmt.setString(3, "bargle");
            pstmt.setInt(4, 1234567890);

            // Execute the prepared statement
            if (pstmt.executeUpdate() == 1)
            {
                System.out.println("Row inserted into database");
            }
            // Close the old prepared statement
            pstmt.close();

            // Create another prepared statement
            pstmt = conn.prepareStatement(
```

```

        "update SSN_Info set ssn=ssn+1 where "+
        "first_name=? and middle_name=? and last_name=?");

// Store the column values for the updated row
pstmt.setString(1, "argle");
pstmt.setString(2, "quinton");
pstmt.setString(3, "bargle");

if (pstmt.executeUpdate() == 1)
{
    System.out.println("The entry has been updated");
}

// Close the old prepared statement
pstmt.close();

// Create another prepared statement
pstmt = conn.prepareStatement(
    "delete from SSN_Info where "+
    "first_name=? and middle_name=? and last_name=?");

// Store the column values for the updated row
pstmt.setString(1, "argle");
pstmt.setString(2, "quinton");
pstmt.setString(3, "bargle");

if (pstmt.executeUpdate() == 1)
{
    System.out.println("The entry has been deleted");
}

    conn.close();
}
catch (Exception exc)
{
    exc.printStackTrace();
}
}
}
}

```

Updating Data from a Result Set

There is another interesting way to update data. You can update items in the result set and then store the updates back into the database. The `ResultSet` interface contains methods for updating items of various data types. The format of the various update methods is similar to the `get` methods, in that the update methods take either a numeric column number or a string column name. The second parameter for each update method is the value you want to store in the column. For instance, to change the value of the `first_name` column, use the following statement:

```
results.updateString("first_name", "MyName");
```

When you create your query statement, you must specify a result set type to let the driver know you want to update result set values. The three statement creation methods—`createStatement`, `prepareStatement`, and `prepareCall`—allow you to specify a result set type and a result set concurrency. The result set type can be `TYPE_FORWARD_ONLY`, `TYPE_SCROLL_INSENSITIVE`, or

TYPE_SCROLL_SENSITIVE. The TYPE_FORWARD_ONLY type indicates that you can only scroll forward through the result set, you can't jump back to a previous result set. For the other two types, you can move to any position in the result set. The sensitive/insensitive variation indicates whether or not the result set is sensitive to external changes to a row.

The options for the concurrency are CONCUR_READ_ONLY and CONCUR_UPDATABLE. If you plan to update rows, add new rows, or delete rows using the result set, you must set the concurrency to CONCUR_UPDATABLE.

Listing 3.3 shows a program that reads rows and updates them using the result set.

Note

The example in Listing 3.3 uses Oracle instead of Cloudscape to use some of the newer JDBC 2.0 features. You will find that not all servers and/or drivers support all the features of JDBC 2.0.

Listing 3.3 Source Code for UpdateResultSet.java

```
package usingj2ee.jdbc;

import java.sql.*;

public class UpdateResultSet
{
    public static void main(String[] args)
    {
        try
        {
            // Make sure the DriverManager knows about the driver
            Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();

            // Create a connection to the database
            Connection conn = DriverManager.getConnection(
                "jdbc:oracle:thin:@flamingo:1521:j2eebook",
                "j2eeuser", "j2eeuser");

            // Create a statement for retrieving and updating data
            Statement stmt = conn.createStatement(
                ResultSet.TYPE_SCROLL_SENSITIVE,
                ResultSet.CONCUR_UPDATABLE);

            stmt.executeQuery("select * from Person");

            // Execute the query
            ResultSet results = stmt.getResultSet();

            while (results.next())
            {
                // Get the name values
                String firstName = results.getString("first_name");
```

```
        String lastName = results.getString("last_name");

// Change the name values
        results.updateString("first_name", firstName.toUpperCase());
        results.updateString("last_name", lastName.toUpperCase());

// Update the row
        results.updateRow();
    }

    conn.close();
}
catch (Exception exc)
{
    exc.printStackTrace();
}
}
}
```

To delete the current row, call the `deleteRow` method:

```
public void deleteRow()
```

To insert a new row, position the result set to a special row called the insert row by calling `moveToInsertRow`:

```
public void moveToInsertRow()
```

You still use the `update` methods to modify the contents for the new row, but when you need to save the changes, call `insertRow`:

```
public void insertRow()
```

The JDBC Optional Package

The Core JDBC package is part of the standard Java Development Kit. There is also an additional JDBC package that is part of J2EE that addresses some of the enterprise-level uses of JDBC. These extensions belong to the `javax.sql` package as opposed to `java.sql`.

Data Sources

One of the irritating parts of JDBC is the way to load drivers. It can be difficult to reconfigure an application to use a different database. Using the standard JDBC API, you must use a specific JDBC URL to access a database. The Optional JDBC package includes an alternative to the `DriverManager` class for creating database connections.

A `DataSource` object works like the `DriverManager` class, it has a `getConnection` method and also a `getLogWriter` method. The big difference is that a data source is typically stored in a naming service and accessed through the Java Naming API (JNDI). The naming service gives you an extra level of indirection—you don't code the JDBC URL into your application, and you don't need to reconfigure every application when you change the database URL. Instead, your application can always ask for a specific name in the naming service. When you configure your application server, you can change the

database that a particular named `DataSource` object refers to. You need to change it in only one place instead of every place that uses the database.

Connection Pools

A *connection pool* is actually just a special data source that maintains a collection of database connections. In a typical application, you need to perform many database operations at once, so you usually need several database connections. The problem is, you don't know when you'll need a particular database connection, so your best bet is to put the connections into a common pool and pull out a connection when you need it.

In the past, programmers have solved the connection pooling problem by creating their own connection pool. One of the problems you often encounter with custom connection pools is that the users of the connections must be extremely polite to each other. If you use a connection from the pool, you can't close the connection, because someone else will need it. You must close all your open statements and result sets, however. Also, you must either commit or rollback any pending database transactions.

Connection pools wrap a special class around a physical database connection. This special class behaves just like a database connection, but you can close the connection and you don't have to worry about any pending transactions. The wrapper class performs any necessary cleanup.

RowSets

A `RowSet` object is similar to a `ResultSet`; in fact, the `RowSet` interface extends `ResultSet`. The `RowSet` interface includes `set` methods in addition to the `get` methods (remember, the `ResultSet` interface uses `update` methods instead of `set`). The `set` methods use only indexed, not column names. The end result is that the `RowSet` is a Java bean with readable and writable properties. The idea is that you can return a `RowSet` to a client object where it can be manipulated and sent back to the server. Also, because a `RowSet` is a bean, you can use it from a GUI design tool.

There isn't anything in the JDBC API that specifies how you get a `RowSet` from a `ResultSet`. Each implementation of `RowSet` must come up with its own way to copy `ResultSet` data into a `RowSet`. Of course, because a `RowSet` is a `ResultSet`, you can use a `RowSet` anywhere you would use a `ResultSet`.

Troubleshooting

Using JDBC

Why does the `DriverManager` tell me that no suitable driver exists?

This usually happens for one of three reasons:

- You mistyped the JDBC URL and the driver manager doesn't recognize the database type.
- You didn't load the JDBC driver class with `Class.forName`, and you didn't add it to the `jdbc.drivers` system property.
- The driver class isn't in the classpath.

My driver is in the classpath; why can't I connect to the database?

Your database URL may be incorrect, or you may be using the wrong driver.

Why do I get errors saying I can't allocate any more database statements after my program has been running for a few minutes?

If you don't explicitly close your `Statement` objects, they don't get closed until the garbage collector picks them up, which may not occur for a long time. Because a database statement takes up resources in the database as well, or at least in the driver, there is usually a limit to the number of open statements you can have. Under a heavy load, you might end up allocating more statements than the database can handle. The garbage collector only understands memory shortages; it isn't smart enough to clean up old statements when the database can't handle any more.

© Copyright Pearson Education. All rights reserved.