Part IV: Java Database Programming

This part of the book discusses how to use Java to develop database projects. You will learn JDBC interfaces and classes, create and process SQL statements, obtaining database metadata, and use JBuilder database beans to rapidly develop Java database applications.

Chapter 12 Introduction to Java Database Programming
Chapter 13 Advanced Java Database Programming
Chapter 14 Applications Using JBuilder DataExpress and dbSwing Components
Introduction to Java Database Programming

Objectives

Understand the architecture of JDBC.

Know the four types of JDBC drivers.

Use JDBC drivers.

Be able to write simple JDBC programs.

Introduction

JDBC is a Java API that provides Java programmers with a uniform interface for accessing and manipulating a wide range of relational databases. Before JDBC, database clients were typically developed using Microsoft Visual Basic, Borland Delphi, and other development tools provided by database vendors, such as PowerBuilder by Sybase and Oracle Developer 2000 by Oracle. These are excellent Rapid Application Development tools, but the programs developed with them can only run on certain platforms. With JDBC, you can write database programs that run on any platform that has a Java Virtual Machine with an appropriate JDBC driver.

The relationship among Java programs, JDBC API, JDBC drivers, and relational databases is shown in Figure 12.1. The JDBC API is a set of Java interfaces and classes used to write Java programs for accessing and manipulating relational databases. Since a JDBC driver serves as the interface to facilitate communications between JDBC and a proprietary database, JDBC drivers are database-specific. You need Oracle JDBC drivers to access Oracle database, and Sybase JDBC drivers to access Sybase database. Even for the same vendor, the drivers may be different for different versions of a database. For instance, the JDBC driver for Oracle 8 is different from the one for Oracle 7. A JDBC-ODBC bridge driver is included in JDK 1.2 to support Java programs that access databases through ODBC drivers. The ODBC driver, introduced before Java, defines a protocol and API for accessing relational databases.

***Same as Fig 12.1 in radjb3, p516***

Figure 12.1

Java programs access and manipulate databases through JDBC drivers.

NOTE: JDBC is the trademarked name of a Java API that supports Java programs to access relational
Overview of the JDBC API

The JDBC API is a Java application programming interface to generic SQL databases that enables Java developers to develop DBMS-independent Java applications using a uniform interface.

The JDBC API consists of classes and interfaces for establishing connections with databases, sending SQL statements to databases, processing the results of the SQL statements, and obtaining database metadata. The JDBC interfaces and classes can be classified into types of driver, connection, statement, result set, metadata, processing support, and exception, as shown in Table 12.1.

NOTE: The JDBC API was initially designed based on JDK 1.02, known as JDBC 1. When JDBC 1 was released, many important Java technologies, such as JavaBeans and Java Internationalization, had not yet been developed. With the release of JDK 1.2 of Java 2, JavaSoft added new features and made some enhancements to JDBC 1. The new JDBC product is known as JDBC 2. JDBC 1 is compatible with Java 2. Any applications developed with JDBC 1 will continue to work under JDK 1.2. However, you will need JDBC 2 drivers to use the new features in JDBC 2. A driver, such as the JDBC-ODBC driver that is bundled with JDK 1.2.x, does not support all features of JDBC 2. If you run the program that uses the unsupported new features, an exception indicating that the operations are not supported would occur. Some of the new features in JDBC 2 are discussed in the section "New Features in JDBC 2," in Chapter 13, "Advanced Java Database Programming."

Table 12.1 JDBC Classes

<table>
<thead>
<tr>
<th>Type</th>
<th>Class/Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Driver management</td>
<td>java.sql.Driver (Interface)</td>
</tr>
<tr>
<td></td>
<td>java.sql.DriverPropertyInfo</td>
</tr>
<tr>
<td></td>
<td>java.sql.DriverManager</td>
</tr>
<tr>
<td>Establishing connections</td>
<td>java.sql.Connection (Interface)</td>
</tr>
<tr>
<td>Processing statements</td>
<td>java.sql.Statement (Interface)</td>
</tr>
<tr>
<td></td>
<td>java.sql.PreparedStatement (Interface)</td>
</tr>
</tbody>
</table>

563
### The interfaces define a framework for generic SQL database access. The JDBC driver vendors provide implementation for these interfaces. The relationship of these interfaces and classes is shown in Figure 12.2. A JDBC application loads an appropriate driver using the Driver interface, connects to the database using the Connection interface, creates and executes SQL statements using the Statement interface, and processes the result using the ResultSet interface if the statements return results. Note that some statements such as SQL data definition statements and SQL data modification statements do not return results.

***Same as Fig 12.2 in radjb3 p518***

### Figure 12.2

JDBC classes enable Java programs to connect to the database, send SQL statements, and process results.

#### The Driver Interface
The Driver interface is database-specific. The JDBC vendor should implement this interface and provide proprietary database information to enable Java programs to communicate with the database.

An appropriate driver must be loaded before your application can connect to a database. The driver can be loaded using the following statement.

```java
Class.forName("TheDriverName")
```

This statement attempts to locate, load, and link the driver class. If it succeeds, it returns an anonymous Class object representing the driver class. You don't need to return an explicit driver reference, because each newly loaded driver is capable of registering itself with the DriverManager.

The driver is anonymous; you don't need to reference it explicitly to make a database connection. After the driver is registered, the DriverManager can use it to make database connections. If the driver is not found, the method throws a ClassNotFoundException.

**The DriverManager Class**

The DriverManager class, a layer of JDBC working between the driver and the user, is responsible for registering and managing the drivers. It serves as a ringmaster to keep track of all the registered drivers, establishing a connection between the database and an appropriate driver.

All the methods in DriverManager are static. A programmer can use its `getConnection()` method to get a connection to the database, which returns an object of Connection. For example, the following statement connects the database at specified url with username and password.

```java
Connection connection =
    DriverManager.getConnection(url, username, password);
```

**The DriverPropertyInfo Class**

The DriverPropertyInfo class is used by advanced programmers to manage specific properties of a Driver instance. It should be used only by developers who need to interact with a driver to discover and supply properties for connections.

**The Connection Interface**

A Connection instance represents a session with a specific database. In the context of a Connection instance, SQL statements are sent to the database for execution, and the results of the execution, if any, are returned. Think of a connection as a
pipeline through which database transactions travel back and forth between your program and the database.

A Connection instance can be used to create various Statement objects for executing SQL statements and stored procedures using the createStatement() method. You can also use it to set transaction properties for the connection. By default the Connection instance automatically commits changes after executing each statement. If auto commit has been disabled, an explicit commit must be done or database changes will not be saved.

You can use the getMetaData() method to get an object of DataBaseMetaData that describes the connected database information on the tables, views, supported SQL grammar, stored procedures, and capabilities of this connection, etc. Chapter 13 gives an in-depth look on retrieving metadata.

The Statement, PreparedStatement, and CallableStatement Interfaces

The Statement, PreparedStatement, and CallableStatement interfaces are related in a hierarchical relationship. CallableStatement inherits PreparedStatement, which inherits Statement. These interfaces are used for sending SQL statements to the database for execution on a given connection. A Statement object is used to execute a static SQL statement; a PreparedStatement object is used to execute a precompiled SQL statement with or without IN parameters; and a CallableStatement object is used to execute a call to a stored procedure in the database.

The executeQuery method of the object is used if the statement is a SQL SELECT query, and the executeUpdate method is used if the statement is a SQL INSERT, UPDATE, or DELETE and also a SQL DDL (Data Definition Language) statement like CREATE TABLE, CREATE VIEW, DROP TABLE, DROP VIEW.

The executeQuery method returns the query result in a ResultSet object. Only one ResultSet per Statement can be open at any point in time. Therefore, if the reading of one ResultSet is interleaved with the reading of another, each must have been generated by different statements. All statement-execute methods implicitly close a statement's current ResultSet if an open one exists.

When executing a stored procedure, the execution may return multiple result sets. The execute method should be used to execute the SQL statement in this case. You can then use getResultSet or getUpdateCount to retrieve the result, and getMoreResults to move to the next result set.

The ResultSet Interface
A ResultSet provides access to a table of data generated by executing a statement.

A ResultSet maintains a cursor pointing to its current row of data. The current row can be set using the cursor movement methods such as absolute, first, last, next, and previous. Initially the cursor is positioned before the first row. The next method moves the cursor to the next row. The first call to next makes the first row the current row, the second call makes the second row the current row, etc.

Various get methods are provided to retrieve column values for the current row. You can retrieve values either by using the index number of the column or by using the name of the column. For example, to retrieve a column of the String type, you may use rs.getString("ColumnName"), where rs is an object of ResultSet.

The column values within a row can be accessed in any order.

The SQL Exception Classes

The JDBC methods may raise the SQLException, SQLWarning, DataTruncation, and BatchUpdateException exceptions. SQLException is the root of all the exceptions in the JDBC.

The SQLException class provides information on a database access error. The SQLWarning exception, a subclass of SQLException, provides information on a database access warning. Warnings are silently chained to the object whose method caused it to be reported. A SQLWarning exception may be ignored in order to allow the normal course of the execution to proceed. A DataTruncation exception, a subclass of SQLWarning, is raised when JDBC unexpectedly truncates a data value. A BatchUpdateException exception, a subclass of SQLException, is raised if exceptions occur during a batch operation.

The DatabaseMetaData Interface

The DatabaseMetaData interface enables you to obtain information about the database, such as tables, views, columns, primary keys, and foreign keys.

Many of these methods return the information in ResultSet. You can use the normal ResultSet methods, such as getString and getInt, to retrieve the data from these result sets. If a given form of metadata is not available, these methods should throw a SQLException. For example, to find all the catalogs in the database, you can use the getCatalogs method to return a list of catalogs in an object of ResultSet, then retrieve all the catalogs from the result set.

The ResultSetMetaData Interface
The ResultSetMetaData interface provides various methods for getting information about a ResultSet object. You can use a ResultSetMetaData object to get the number of columns and find out about the types and properties of the columns in a ResultSet.

The JDBC Support Classes

JDBC provides many support classes. The Date, Time, and Timestamp classes are for accepting and processing SQL DATE, TIME, and TIMESTAMP values. All these classes extend the java.util.Date class.

The Types class contains a list of predefined constants for identifying SQL types that can be used in JDBC such as INTEGER, FLOAT, DOUBLE, VARCHAR, BLOB, CLOB, etc. No methods are defined in this class.

The Blob, Clob, Array, Ref, Struct, SQLInput, and SQLOutput are the new support classes introduced in JDBC 2. The Blob class represents a SQL BLOB type for storing binary large object such as an image. The Clob class represents a SQL CLOB type for storing character large object such as a data file. The Array class represents a SQL ARRAY type for storing a sequence of values. The Ref class is a reference to a SQL structured type value in the database. The Struct class maps to a SQL structured type. The SQLInput and SQLOutput classes provide a mechanism for mapping structured SQL types to Java objects.

Developing JDBC Applications

The JDBC interfaces and classes are the building blocks in the development of Java database programs. A typical Java program takes the following steps to access the database, as shown in Figure 12.2.

1. Load drivers.

JDBC allows multiple drivers to be loaded in one program. A program can load a JDBC driver at any time. For example, MyDriver is loaded with the following statement:

```java
Class.forName("MyDriver");
```

The statement loads the driver. The driver is automatically registered with the driver manager. You may use the static method registerDriver in the DriverManager class to explicitly register a driver with the driver manager. For example, the following statement registers MyDriver.

```java
DriverManager.registerDriver(new MyDriver());
```

2. Establish connections.
Once the driver is loaded and registered with the driver manager, the `DriverManager` is capable of choosing an appropriate one from a list of registered drivers to fulfill the connection to the database. You simply invoke `DriverManager.getConnection()` to establish a session with the database. In the rare case that more than one driver is suitable to connect to a database and you want a particular driver to be chosen, the `connect` method in a `Driver` object can be invoked directly to use that driver to connect to a given database.

3. Create statements.

You can create statements for executing SQL in three types: `Statement`, `PreparedStatement`, or `CallableStatement`. The `Statement` class can be used to execute static SQL. The `PreparedStatement` and `CallableStatement` classes are used to construct dynamic SQL.

4. Execute statements.

The statements can be executed using the methods `executeQuery`, `executeUpdate`, or `execute`. The result of the query is returned in `ResultSet`.

5. Process `ResultSet`.

The `ResultSet` maintains a table whose current row can be retrieved. The initial row position is null. The methods such as `absolute`, `first`, `last`, `next`, and `previous` can be used to set the row position. You can use various get methods to retrieve values from a current row.

Here is a simple example (see Listing 12.1) of connecting to a database, executing a simple query, and processing the query result.

Listing 12.1: SimpleJDBC.java

```java
package jdbcdemo;

import java.sql.*;

public class SimpleJDBC {
    public static void main(String[] args)
        throws SQLException, ClassNotFoundException {
        // Load the JDBC driver
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        System.out.println("Driver loaded");

        // Establish a connection
        Connection connection = DriverManager.getConnection
            ("jdbc:odbc:LiangBookDB_MDB", ",");
        System.out.println("Database connected");
```
// Create a statement
Statement stmt = connection.createStatement();

// Select the columns from the Student table
ResultSet rset = stmt.executeQuery
("select firstName, mi, lastName from Student");

// Iterate through the result and print the student names
while (rset.next())
    System.out.println(rset.getString(1) + " " + rset.getString(2) + ". " + rset.getString(3));

The statement

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

loads the JdbcOdbcDriver for connecting to an ODBC data source, such as Microsoft Access Database. In Example 12.1, you will learn how to set up the ODBC data source to test this program.

The statement

Connection connection = DriverManager.getConnection
("jdbc:odbc:LiangBookDB_MDB", ", ");

creates a Connection object connection, which represents a logical session for sending SQL statements and getting query results back from the database. You can create an instance of Statement for executing static queries as used in the example, create PreparedStatement for precompiled SQL statements, or create CallableStatement for executing stored procedures. Their applications are discussed in Chapter 13.

The statement

ResultSet rset = stmt.executeQuery
("select firstName, mi, lastName from Student");

executes the query and returns the result in a ResultSet object, which can be perceived as a table of three columns consisting of firstName, mi, and lastName. The getString(1), getString(2), and getString(3) methods retrieve the column values for firstName, mi, and lastName, respectively. Alternatively, you can use getString("firstName"), getString("mi"), and getString("lastName") to retrieve the same three column values.

The first execution of the next() method sets the current row to the first row in the table, and subsequent invocation of the next() method sets the current row to the second, third, and so on, to the last row.
NOTE: The MS Access database (LiangBook.mdb) in the book is created using Microsoft Access 2000. This database cannot be used with MS Access 97. If you use MS Access 97 or 95, please create your own database.

**JDBC Drivers**

A JDBC driver is a software component that works between JDBC and the database. The driver is responsible for accepting SQL statements, sending them to the database system, and returning the execution results to JDBC. The driver is specific to each database. For instance, to enable JDBC to connect to the Oracle database, you need a JDBC driver that works specifically for Oracle. Even among Oracle databases, the drivers are different for various versions of Oracle DBMS.

There are many JDBC drivers now available on the market. They can be classified into the four types listed in Table 12.2.

**Table 12.2 JDBC Driver Types**

<table>
<thead>
<tr>
<th>Types of drivers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type 1: JDBC-ODBC Bridge</td>
<td>Provides access through ODBC drivers</td>
</tr>
<tr>
<td>Type 2: Native-API</td>
<td>Provides access through native database API</td>
</tr>
<tr>
<td>Type 3: Middle-tier</td>
<td>Provides access through a middle-tier server</td>
</tr>
<tr>
<td>Type 4: Native-protocol</td>
<td>Provides access through a network protocol used by DBMS</td>
</tr>
</tbody>
</table>

**Type 1: JDBC-ODBC Bridge Driver**

ODBC (Open DataBase Connectivity) is an API for accessing relational databases. It was originally designed to enable database clients on PCs to access databases on a server by Microsoft. It became a popular industry standard API for connecting to the relational databases for the client on all types of platforms. Since ODBC is widely available, it makes sense to provide a driver that accesses the database through ODBC drivers. Such a driver is referred to as a *JDBC-ODBC bridge driver*.

A JDBC-ODBC driver is bundled with JDK 1.1 and JDK 1.2, which is a joint product by Sun and Intersolv. This driver bridges JDBC with ODBC by mapping the JDBC calls into ODBC calls, and the ODBC calls interact with the database, as shown in Figure 12.3.
The JDBC-ODBC bridge driver maps JDBC calls to ODBC calls. The JDBC-ODBC bridge driver provides convenient access to all the existing systems, since most of the database systems already have ODBC drivers. However, the driver is slow and is not suitable for applets. The ODBC drivers are specific to the database. For example, you need an Oracle ODBC driver to access Oracle databases. The JDBC application using the JDBC-ODBC bridge driver requires that the ODBC driver and the client database library must be loaded on the client machine. This approach is obviously not efficient, since it requires the JDBC driver to talk to the database via an ODBC driver. Since the ODBC drives is written in C, the driver cannot be downloaded and used by applets.

Type 2: Native-API Driver

This kind of driver converts JDBC calls into native database calls and communicates directly with the database server, as shown in Figure 12.4.

Native-API drivers are usually provided by database vendors. For example, Oracle provides a native-API driver that maps JDBC with the Oracle Call Interface. Like the bridge driver, this driver requires some binary code for the native database API to be loaded on the client machine. But it is more efficient than the JDBC-ODBC bridge approach, since it allows you to talk to the database directly.

Type 3: Middle-Tier Driver

This kind of driver converts JDBC calls into a DBMS-independent network protocol and sends them to a middle-tier server. The server then translates the calls to a DBMS-specific protocol for Oracle, Sybase, Informix, InterBase, or other DBMS, as shown in Figure 12.5.
The middle-tier JDBC server connects to Java applications or applets with a database server. Inprise's InterClient and Symantec's dbAnyWhere are middle-tier servers based on this kind of architecture. The middle-tier server can reside on a machine different from the client and the DBMS server.

The middle-tier driver is a pure Java JDBC driver that enables platform-independent, client/server development for the Internet. The advantage of a pure Java driver versus a native-code driver is that you can deploy applets without having to manually load platform-specific JDBC drivers on each client system. Since the Web servers automatically download the driver’s classes along with the applets, there is no need to manage local native database libraries, and this simplifies administration and maintenance of customer applications. As part of a Java applet, the driver can be dynamically updated, further reducing the cost of application deployment and maintenance.

An example of using the middle-tier driver with Borland’s InterClient and InterBase database can be obtained from www.cs.armstrong.edu/liang/radjb5.html.

**Type 4: Native-Protocol Driver**

The native-protocol driver converts JDBC calls into network protocol used by a DBMS and communicates with the DBMS directly, as shown in Figure 12.6.

***Same as Fig 12.6 in radjb3 p526***

**Figure 12.6**

The native-protocol driver facilitates communications between Java applications and applets with a DBMS server.

Like the middle-tier driver, this kind of driver does not require any database specific binary code to be loaded on the client. The native-protocol driver, however, is more flexible than the middle-tier driver, because it eliminates the middle-tier JDBC server.

NOTE: The JDBC-ODBC driver and the native-API driver use native methods, which makes them platform-specific. They are not suitable for Java applets intended to be downloaded into browsers running on universal platforms. Middle-tier drivers are preferred for use with multi-tier architecture where connectors are used to link clients with database servers. Native-protocol drivers are ideal for direct access to the database. Both middle-tier and native-protocol drivers are written in 100% pure Java; thus they are suitable for Java applets.
**Connecting to Databases Using JDBC Drivers**

A number of JDBC drivers have already been developed. Many of them can be downloaded from vendors for 30 or more days of evaluation. Information for locating the drivers can be obtained from the following URL.

http://industry.java.sun.com/products/jdbc/drivers

This section provides examples of the four types of JDBC drivers. Specifically, you will use the Sun JDBC-ODBC driver, the Oracle JDBC OCI driver (native-API), the Inprise InterClient driver (middle-tier), and the Oracle thin-client driver (native-protocol).

NOTE: A driver is not just of a single class. In fact, a JDBC driver is a set of classes that provide mappings between JDBC and a specific database. The driver must include classes that implement the eight interfaces in the java.sql package: Driver, Connection, Statement, PreparedStatement, CallableStatement, ResultSet, DatabaseMetaData, and ResultSetMetaData. For example, the oracle.jdbc.driver.OracleStatement class implements java.sql.Statement. Figure 12.7 lists all the classes in the Oracle 8i JDBC Thin driver bundled in the classes12.zip file using the WinZip utility. These classes are used by JDBC, not directly by JDBC programmers.

![WinZip (Unregistered) - classes12.zip](image)

**Figure 12.7**

A JDBC driver contains classes that enable JDBC to interact with a specific database.
JDBC URL Naming Conventions

How does a Java program find its database? It finds the database by using its JDBC URL. A JDBC URL provides a way for the appropriate driver to identify the database. The standard syntax of a URL is:

```java
jdbc:<subprotocol>:<datasource>
```

The first part of the syntax is always `jdbc`. The second part is the `subprotocol` specified by the driver vendor. For example, the subprotocol for Sun drivers is always `sun`, and the subprotocol for Oracle drivers is always `oracle`. The `datasource` consists of the database name and the host machine where the database resides. Information needed to access the data source, such as the user's login name and password, may be included in `datasource`, or may be supplied separately.

The subprotocol is used by the `DriverManager` class to associate an appropriate driver with the database. When the `DriverManager` class presents the `subprotocol` of the URL to its list of registered drivers, the driver that matches the protocol is used to establish a connection to the database specified in the data source. The subprotocol name is defined by the vendor, but it must be registered with JavaSoft to avoid naming conflicts. JavaSoft acts as an informal registry for JDBC subprotocol names.

The subprotocol `odbc` is reserved for JDBC-ODBC bridge drivers. For example, the URL to access an ODBC data source "MyDataSource" through a JDBC-ODBC driver is

```java
jdbc:odbc:MyDataSource
```

In this example, the `subprotocol` is `odbc`, and the data source is the ODBC data source "MyDataSource."

The subprotocol may consist of several words. For example, the subprotocol `oracle:oci8` is reserved for accessing Oracle 8 database through native Oracle OCI (Oracle Call Interface), and `oracle:thin` is reserved for accessing oracle 8 using the Type 4 native-protocol driver.

The JDBC URL provides an indirection in the data source. You may use a logical host or database name that is dynamically translated to an actual name by a network naming system. For example, the URL to access a remote Oracle 8 database using the Oracle JDBC Thin driver may look like:

```java
jdbc:oracle:thin:scott/tiger@liang.armstrong.edu:1521:test
```
This URL implies that the Oracle database with the ID oracle, runs at the port 1521 on liang.armstrong.edu. The user name is scott and the password is tiger.

**Using the JDBC-ODBC Driver**

To use the JDBC-ODBC driver to access databases in Java, two drivers must be installed on the client machine: a universal JDBC-ODBC bridge driver and a vendor-specific ODBC driver. The JDBC-ODBC driver comes with Java 2 SDK 1.3 or higher, or any Java development tool that supports JDK 1.2, or higher. The ODBC driver is not included in the JDK and is usually provided by database vendors. The example chosen to demonstrate the JDBC-ODBC approach in this section is the Microsoft ODBC driver for the MS-Access database on Windows, since it is bundled with Microsoft Office Suite and widely available to many readers. This section uses the ODBC driver on Windows 2000.

NOTE: Using the JDBC-ODBC approach, the database is not necessarily on the same machine with the client. For example, you may use an Oracle ODBC driver to access a remote Oracle database.

**Example 12.1 Using the JDBC-ODBC Bridge Driver**

This example develops a Java program that creates a table named Student in an MS Access database. Each record consists of student firstName, mi, and lastName. Your program inserts two records into Student and then displays all the records on the console.

Follow the steps below to complete the project.

1. Install an MS ODBC driver for MS Access if necessary.

By default the ODBC driver is installed on Windows 2000. If not, install MS Access to get the proper ODBC driver on your system. Upon successful installation, you should see the icon Data Sources (ODBC) in the Administrative Tools subwindow under control panel for Windows 2000. For Windows 98 and NT, you should see a "32-bit ODBC" icon appearing on the control panel.

2. Set up a Data Source as the database for storing the Student table.

2.1. From the Windows Start button, choose Setting, Control Panel to bring up the Control Panel dialog box.

2.2. Double-click Administrative Tools, and then double-click Data Sources (ODBC) to display the "ODBC Data Source Administrator," as shown in Figure 12.8.
2.3. Click Add to bring up the "Create New Data Source" dialog box, as shown in Figure 12.9.

2.4. Select Microsoft Access Driver and Press "Finish" to bring the "ODBC Microsoft Access Setup" dialog window, as shown in Figure 12.10. Type "LiangBookDB_MDB" in the Data Source Name field, and type "Database Text MS Access database" in the Description field. Press Select to bring up the "Select Database" dialog window, as shown in Figure 12.11.

2.5. Since LiangBookDB_MDB.mdb was already created, select LiangBookDB_MDB.mdb from the directory c:\LiangBookDB_MDB. Press OK to close the Select Database dialog window, click OK to close the ODBC Microsoft Access Setup window, and click OK to close the ODBC Data Source Administrator window.

Figure 12.8

The ODBC Data Source Administrator is the main dialog box to manage the data source and the drivers.
Figure 12.9

Select a proper driver for the Data Source in the "Create New Data Source" window.

Figure 12.10

Specify the Data Source Name to associate it with a database in the "ODBC Microsoft Access Setup" window.
3. Create a new project named jdbcdemo.jpr and create a class named TestJdbcOdbc (see Listing 12.2) to create a Student table, insert two records, and retrieve the records from the database.

Listing 12.2: TestJdbcOdbc.java

```java
package jdbcdemo;

import java.sql.*;

public class TestJdbcOdbc {
    public static void main(String[] args)
            throws SQLException, ClassNotFoundException {
        // Load the JDBC-ODBC bridge driver
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        System.out.println("Driver sun.jdbc.odbc.JdbcOdbcDriverloaded");

        // Establish connection
        Connection connection = DriverManager.getConnection
                (
                "jdbc:odbc:LiangBookDB_MDB", ", ");
        System.out.println("Database LiangBookDB_MDB connected");

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

        try {
            // Create the Student TABLE
            stmt.executeUpdate("CREATE TABLE Student (firstName varchar(20), mi varchar(1), lastName varchar(20))");

            // Insert two records into the Student TABLE
```
stmt.executeUpdate("insert into Student values ('John', 'F', 'Smith')");
stmt.executeUpdate("insert into Student values ('Wade', 'E', 'Bush')");

} catch (SQLException ex) {
    System.out.println("Table Student already exists");
}

// Select the column student from the Student table
ResultSet rset = stmt.executeQuery("select firstName, mi, lastName from Student");

// Iterate through the result and print the student names
while (rset.next())
    System.out.println(rset.getString(1) + " " + rset.getString(2) + ". " + rset.getString(3));

// Close the connection
connection.close();

4. Compile and test the program. A sample run of the program is shown in Figure 12.12.

![Command Prompt]

C:\radjb>java jdbcdemo.TestJdbcOdbc
Driver sun.jdbc.odbc.JdbcOdbcDriverloaded
Database LiangBookDB_MDB connected
John F. Smith
Wade E. Bush

C:\radjb>java jdbcdemo.TestJdbcOdbc
Driver sun.jdbc.odbc.JdbcOdbcDriverloaded
Database LiangBookDB_MDB connected
Table Student already exists
John F. Smith
Wade E. Bush

Figure 12.12

The Java program accesses an MS Access data source through the JDBC-ODBC bridge driver.

Example Review

Java does not run on 16-bit platforms. You must use a 32-bit ODBC driver, not a 16-bit ODBC driver. The program
was tested using MS Access 2000. It should also work with MS Access 97.

The URL of a data source for a JDBC-ODBC connection is 
jdbc:odbc:ODBC-Source-Name. So jdbc:odbc:LiangBookDB_MDB 
is the URL used in the statement

```java
Connection connection = DriverManager.getConnection
   ("jdbc:odbc:LiangBookDB_MDB", ",", ");
```

to establish a connection to the data source 
LiangBookDB_MDB.

The program creates the Student table, inserts two 
records, and displays the records in the table. The SQL 
statements are passed as string arguments.

**TIP:** The statement `connection.close()` closes the 
connection. This is statement is not necessary in 
this example because the connection is automatically 
closed upon exiting the program. However, I recommend 
you to use this statement whenever a connection is no 
longer needed. Closing it would release the resource 
associated with the connection.

**NOTE:** When executing this example using MS Access 97, 
close MS Access, because only one active session is 
allowed on MS Access 97. Access Windows 2000 running 
on Windows 2000 does not have this restriction.

### Using the Native-Protocol Driver

A native-protocol driver translates JDBC calls into network 
protocol used by a DBMS and communicates with the DBMS directly. 
Since some part of the network protocol for facilitating database 
communications is vendor-specific, a driver of this kind is 
usually provided by the vendor. The driver is written in 100%
pure Java to implement high-level network communication to the 
server. Since the native-protocol driver, unlike the native-API 
driver, does not require that any database specific binary code 
be loaded on the client, it is suitable for deploying Java 
applets.

Oracle, the most popular database, is used by many large 
corporations. In this section, you will learn how to use the 
Oracle JDBC thin driver to develop Java applets for accessing 
remote Oracle databases. Oracle's JDBC thin driver is a native-
protocol driver that provides an implementation of the Oracle 
high-level communications protocol known as TNS (Transparent 
Network Substrate). This driver can access local or remote 
servers of Oracle 8i or higher from a client on any platform.
NOTE: You have to use a correct version of Oracle driver for the database. The Oracle JDBC OCI drivers for Oracle 7 and Oracle 8 are different.

**Example 12.2 Using the Oracle JDBC Thin Driver**

This example develops a Java applet that connects to a remote Oracle 8i server using an Oracle JDBC Thin driver. After the connection is established, the program obtains the system time from the server and displays it on the applet.

Follow the steps below to complete the project.

1. If you have already installed Oracle 8i, it is likely you already have the JDBC driver installed in \jdbc\lib of your Oracle installation directory. If not, join Oracle Tech Network to download it. It is bundled with Personal Oracle 8i.

2. In the jdbcdemo project, create an applet named TestOraJdbcThin that connects to the test database on liang.armstrong.edu, obtains current system time from the server, and displays it on the applet. The program is given in Listing 12.3.

3. Add \jdbc\lib\classes12.zip in the libraries in the Project Properties in JBuilder if you use JBuilder. Add \jdbc\lib\classes12.zip in the classpath if you run the program from the DOS prompt. Compile TestOraJdbcThin.java and run the program. A sample run of the program is shown in Figure 12.13.

Listing 12.3: TestOraJdbcThin.java

```java
package jdbcdemo;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import javax.swing.*;
import java.sql.*;

public class TestOraJdbcThin extends JApplet {
    // SQL Statement
    Statement stmt;
    boolean isStandalone = false;
    JPanel jPanel1 = new JPanel();
    JTextArea jtaOutput = new JTextArea();
    BorderLayout borderLayout1 = new BorderLayout();
    JLabel jLabel1 = new JLabel();
    JTextField jtfTime = new JTextField();
    JButton jbtGetTime = new JButton();
}
```
/**Initialize the applet*/
public void init() {
    try {
        jbInit();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

/**Component initialization*/
private void jbInit() throws Exception {
    this.setSize(new Dimension(399, 166));
    jPanel1.setLayout(borderLayout1);
    JLabel1.setText("Server Time");
    jbtGetTime.setText("Get Server Time");
    jbtGetTime.addActionListener(
        new java.awt.event.ActionListener() {
            public void actionPerformed(ActionEvent e) {
                jbtGetTime_actionPerformed(e);
            }
        });
    this.getContentPane().add(jPanel1, BorderLayout.NORTH);
    jPanel1.add(jLabel1, BorderLayout.WEST);
    jPanel1.add(jtfTime, BorderLayout.CENTER);
    jPanel1.add(jbtGetTime, BorderLayout.EAST);
    this.getContentPane().add(jtaOutput, BorderLayout.CENTER);
    // Initialized JDBC
    initializeJDBC();
}

private void initializeJDBC() {
    try {
        // Load the Oracle JDBC Thin driver
        Class.forName("oracle.jdbc.driver.OracleDriver");
        jtaOutput.append("Driver oracle.jdbc.driver.OracleDriver" +
            " loaded"+'\n');

        // Connect to the sample database
        Connection connection = DriverManager.getConnection
            ("jdbc:oracle:thin:@liang.armstrong.edu:1521:test",
             "scott", "tiger");
        jtaOutput.append("Database jdbc:oracle:thin:scott/tiger" +
            "@liang.armstrong.edu:1521:test connected"+'\n');

        // Create a Statement
        stmt = connection.createStatement();
    } catch (Exception ex) {
        jtaOutput.append(ex.getMessage() + '\n');
    }
}
void jbtGetTime_actionPerformed(ActionEvent e) {
    String query = "select sysdate from Dual";

    if (e.getSource() instanceof JButton) {
        try {
            // Execute the query
            jtaOutput.append("Executing query " + query + 
"\n");
            ResultSet rset = stmt.executeQuery(query);

            // Display the time
            while (rset.next())
                jtfTime.setText(rset.getString(1));
        }
        catch (Exception ex) {
            jtaOutput.append(ex.getMessage() + '\n');
        }
    }
}

Figure 12.13

Using the Oracle JDBC Thin driver, the Java applet accesses a remote database and obtains the time from the server.

Example Review

The Oracle JDBC Thin connection URL is
jdbc:oracle:thin:@hostname:port:sid. So the following statement

Connection conn = DriverManager.getConnection
("jdbc:oracle:thin:@liang.armstrong.edu:1521:test",
"scott", "tiger");

establishes a connection to the remote database on liang.armstrong.edu using username scott and password tiger to the database with a system id test whose listener is at port 1521.
You can test the applet using the appletviewer utility, as shown in Figure 12.14. This applet can be deployed on the server where the database is located so that any client on the Internet can run it from a Web browser. Since the client may not have the Oracle Thin driver, you should make the driver available along with the applet in one archive file. This archive file can be created as follows:

1. Copy \jdbc\lib\classes12.zip to TestOraJdbcThin.zip.

2. Add TestOraJdbcThin.class into TestOraJdbcThin.zip using the WinZip utility.

3. Add TestOraJdbcThin$1.class into TestOraJdbcThin.zip using the WinZip utility. TestOraJdbcThin$1.class is for the anonymous inner event adapter class for listening to the button action.

You need to deploy TestOraJdbcThin.zip and TestOraJdbcThin.html on the server. TestOraJdbcThin.html should use the applet tag with a reference to the Zip file as follows:

```html
<applet
   code="TestOraJdbcThin"
   archive="TestOraJdbcThin.zip"
   width=500
   height=200
>
</applet>
```

NOTE: You can also create the archive file using the Archive Builder. Be careful, however, the Archive Builder wizard is not very intelligent. It does not package the Oracle driver classes to the JAR file, unless you check the option “Include all classes and resources” in Step 3 of 5 of the Archive Builder. With this option checked, the Archive Builder packages all the classes and files in the project, though many of the files are not needed for this applet. To circumvent the problem, you may create a new project that contains just classes for the applet and its dependent files, then package all the classes in this project into one archive file.

NOTE: To access the database from an applet, it is necessary, because of security restrictions, for the applet to be downloaded from the server where the Oracle database is located. Hence, you have to deploy the applet on the server.

NOTE: Do not use semicolon (;) to end the Oracle SQL command. The semicolon does not work with the Oracle JDBC drivers. The semicolon works with other drivers used in the book.
Using the Native-API Driver

A native-API driver translates JDBC calls directly into calls on the DBMS-specific client API. Thus, you must install the JDBC driver as well as the DBMS client-side library. Major enterprise database vendors like Oracle, Sybase, and Informix already provide native support of database connectivity for the development tools, such as Visual Basic and Borland Delphi, among many others, through their proprietary client API. The support extended by these companies enables JDBC to talk directly to the client API through the native-API driver.

This section demonstrates developing a Java application that accesses a remote Oracle database via the Oracle JDBC OCI driver. Oracle's JDBC OCI driver is a native-API driver that provides an implementation of the JDBC interfaces on top of the OCI (Oracle Call Interface) to interact with an Oracle database. This driver can access local or remote Oracle database from a client on Windows.

Example 12.3 Using the Oracle JDBC OCI Driver

This example develops a Java program that simply gets the system time from a remote database. The server is accessible on the Internet with the host name liang.armstrong.edu (IP address 130.254.204.33). An Oracle 8i database is running on the server. You can access the database using username scott and password tiger.

Follow the steps below to set up the client on Windows.

1. Install Oracle 8i client if necessary. Oracle 8i client includes SQL*Net and all other dependent files required to support JDBC OCI driver to connect to the Oracle server. SQL*NET is an Oracle proprietary product that supports network service to the server from the client.

2. Install the Oracle 8i JDBC driver if necessary. Your Oracle client CDROM should contain the driver. The driver is a compressed .zip file. There are several .zip files. This example uses the classes12.zip file to connect to Oracle 8i with Java 2. After installation, the driver files are in the \jdbc\lib folder under the Oracle installation directory.

3. Add the Oracle TNS alias name for the sample database in the \network\admin\tnsname.ora file as follows, so that this program can access my Oracle test database for this book.

liang -
4. In the jdbcdemo project, create a class name TestOraJdbcOci8 as shown in Listing 12.4.

5. To run the program from the DOS prompt, you need to add \jdbc\lib in the path, and add \jdbc\lib\classes12.zip in the classpath. To run the program from JBuilder, you need to add \jdbc\lib\classes12.zip in the Java libraries in the Project Properties.

Listing 12.4: TestOraJdbcOCI8.java

```java
package jdbcdemo;

import java.sql.*;

public class TestOraJdbcOci8 {
    public static void main(String[] args)
        throws SQLException, ClassNotFoundException {
        // Load the Oracle JDBC driver
        Class.forName("oracle.jdbc.driver.OracleDriver");
        System.out.println("Driver loaded");

        // Connect to the sample database
        Connection connection = DriverManager.getConnection
            ("jdbc:oracle:oci8:@liang", "scott", "tiger");
        System.out.println("Database connected");

        // Create a Statement
        Statement stmt = connection.createStatement();

        // Select the date and time from the server
        ResultSet rset = stmt.executeQuery("select sysdate from Dual");

        // Show the system date and time
        while (rset.next())
            System.out.println(rset.getString(1));
    }
}
```
6. Compile and run the program. A sample run of the program is shown in Figure 12.14.

![Command Prompt](image)

**Figure 12.14**

Using an Oracle 8 JDBC OCI driver, the Java program accesses a remote database and obtains the time from the server.

**Example Review**

The Oracle 8 JDBC OCI connection url is `jdbc:oracle:oci8@tnsname`. So the following statement

```java
Connection connection = DriverManager.getConnection(
    "jdbc:oracle:oci8:@liang", "scott", "tiger");
```

establishes a connection to the remote database on the host liang.Armstrong.edu through the TNS alias name liang using username `scott` and password `tiger`.

The SQL statement "select sysdate from Dual" is specific to Oracle, which returns the current system date and time on the database server.

The success of this example depends on a sequence of steps, each of which must be done exactly right to avoid frustrating errors. Here is a list of things to check:

[BL] Check whether you can log on to the database using `sqlplus scott/tiger@liang`. If failed, check whether you have created a right alias in the tnsname.ora file for the database server.

[BL] If you get a `ClassNotFoundException`, it is likely that you did not set `\jdbc\lib\classes12.zip` in the classpath.

[BL] If you get an error indicating that ocijdbc8.dll file not found when you run this program from the DOS prompt, it is because you did not add `\jdbc\bin` in the path.
A Universal SQL Client

You have learned to use various drivers to connect to the database, to create statements for executing SQL statements, and to process the results from SQL queries. This section presents a bean component for connecting to databases, and a useful utility for processing interactive SQL commands.

**Example 12.4 Creating a Connection Dialog Component**

This example develops a dialog bean component to connect to a database, as shown in Figure 12.15. The user enters or selects a JDBC driver, a database URL, a username, and a password, and clicks the Connect button to connect to the database. Upon connected, an instance of the Connection interface can be obtained from the getConnection method in the dialog bean.

![Figure 12.15](image)

The Connect to a database dialog box enables to connect to any JDBC data source.

Here are the steps to create this component.

1. In the jdbcdemo.jpr project, create a dialog class named DBConnectDialog using the Dialog wizard from the Object Gallery. The Dialog wizard creates a panel named panel1 of BorderLayout in the dialog container.

2. In the UI Designer for DBConnectDialog.java, set the modal property of the dialog to true, and the title to “Connect to a database.” Drop a JPanel to the center of panel1 to create jPanel1, and drop a JPanel to the south of panel1 to create jPanel2.

3. In jPanel1, set the layout to BorderLayout, and choose titled in the border property to create titledBorder1. Set the title property of titledBorder1 to “Enter database information” in the property editor for titleBorder1.
4. Drop a JPanel to the west of JPanel1 to create JPanel3, and drop a JPanel to the center of JPanel1 to create JPanel4. Set the layout of both JPanel3 and JPanel4 to GridLayout, with the row property set to 4 and the column property set to 1.

5. Drop a JLabel four times to JPanel3, and set the text of the labels to "JDBC Driver," "Database URL," "Username," and "Password." Drop a JComboBox twice to JPanel4, and rename the combo boxes j cboDriver, and j cboURL. Set the editable property of the combo boxes to true so to allow the user to type in a driver name and URL in addition to select them from the combo boxes. Drop a JTextField to JPanel4 and rename the object jtfUsername. Drop a JPasswordField to JPanel4 and rename the object jpfPassword.

6. Set the layout of JPanel2 to FlowLayout with right alignment. Drop a JButton twice to create buttons Connect and Cancel, and rename the buttons jbtConnect and jbtCancel.

7. Click the Bean tab in the Content pane to switch to BeansExpress. Choose the Properties tab and click the Add Properties button to add a property named connection of the java.sql.Connection type with a get method. This field stores the connection to the database once a connected session is established.

8. Use the addItem method to add JDBC driver classes, and database URLs to j cboDriver and j cboURL. Generate and implement the handlers for the Connect and Cancel buttons, as shown in Example 12.5.

Example 12.5: DBConnectDialog.java

```java
package jdbcdemo;

import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;
import java.awt.event.*;
import java.sql.*;

public class DBConnectDialog extends JDialog {
    // Generated from BeansExpress
    private java.sql.Connection connection;
    JPanel panel1 = new JPanel();
    BorderLayout borderLayout1 = new BorderLayout();
    JPanel jPanel1 = new JPanel();
    JPanel jPanel2 = new JPanel();
    FlowLayout flowLayout1 = new FlowLayout();
    ```

590
**Construct a dialog with specified frame, title, and modal*/
public DBConnectDialog(Frame frame, String title, boolean modal) {
    super(frame, title, modal);
    try {
        jbInit();
        pack();
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}

/**Default constructor*/
public DBConnectDialog() {
    this(null, "", false);
}

void jbInit() throws Exception {
    titledBorder1 = new TitledBorder("");
    JPanel jPanel1 = new JPanel();
    JPanel jPanel2 = new JPanel();
    GridLayout gridLayout1 = new GridLayout();
    JLabel jLabel1 = new JLabel();
    JLabel jLabel2 = new JLabel();
    JLabel jLabel3 = new JLabel();
    JLabel jLabel4 = new JLabel();
    JComboBox jcboDriver = new JComboBox();
    JComboBox jcboURL = new JComboBox();
    JTextField jtfUsername = new JTextField();
    JPasswordField jpfPassword = new JPasswordField();

    jPanel1.setBorder(titledBorder1);
    jPanel2.setLayout(flowLayout1);
    flowLayout1.setAlignment(2);
    jbtConnect.setText("Connect");
    jbtConnect.addActionListener(
        new java.awt.event.ActionListener() {
            public void actionPerformed(ActionEvent e) {
                jbtConnect_actionPerformed(e);
            }
        });
    jbtCancel.setText("Cancel");
    jbtCancel.addActionListener(
        new java.awt.event.ActionListener() {
            public void actionPerformed(ActionEvent e) {
                jbtCancel_actionPerformed(e);
            }
        });
    jPanel1.setBorder(titledBorder1);
```java
// JPanel1.setLayout(gridLayout2);
// // titledBorder1.setTitle("Enter Database Information");
// JPanel3.setLayout(gridLayout1);
// JLabel1.setText("JDBC Driver");
// JLabel2.setText("Database URL");
// JLabel3.setText("Username");
// JLabel4.setText("Password");
// gridLayout1.setColumns(1);
// gridLayout1.setRows(4);
// JPanel4.setLayout(gridLayout2);
// gridLayout2.setColumns(1);
// gridLayout2.setHgap(10);
// gridLayout2.setRows(4);
// this.setTitle("Connect to a database");
// this.setModal(true);
// jcbDriver.setEditable(true);
// jcbURL.setEditable(true);
// getContentPane().add(panel1);
// panel1.add(jPanel1, BorderLayout.CENTER);
// jPanel1.add(jPanel3, BorderLayout.WEST);
// jPanel3.add(jLabel1, null);
// jPanel3.add(jLabel2, null);
// jPanel3.add(jLabel3, null);
// jPanel3.add(jLabel4, null);
// jPanel1.add(jPanel4, BorderLayout.CENTER);
// jPanel4.add(jcbDriver, null);
// jPanel4.add(jcbURL, null);
// jPanel4.add(jtfUsername, null);
// jPanel4.add(jpfPassword, null);
// panel1.add(jPanel2, BorderLayout.SOUTH);
// jPanel2.add(jbtConnect, null);
// jPanel2.add(jbtCancel, null);

// // Add items to the combo boxes
// jcbDriver.addItem("sun.jdbc.odbc.JdbcOdbcDriver");
// jcbURL.addItem("jdbc:odbc:LiangBookDB_MDB");
// jcbDriver.addItem("oracle.jdbc.driver.OracleDriver");
// jcbURL.addItem("jdbc:oracle:thin:@liang.armstrong.edu:1521:test");
// // Oracle Type 2 JDBC Connection
// jcbURL.addItem("jdbc:oracle:oci8:LiAng");
// jcbDriver.addItem("com.borland.datastore.jdbc.DataStoreDriver");
// jcbURL.addItem("jdbc:borland:dslocal:C:\radjb\LiangBookDB.Jds");
// jcbDriver.addItem("interbase.interclient.Driver");
// jcbURL.addItem("jdbc:interbase://localhost/C:/LiangBook/LiangBookDB.gdb");

// // Username and password for Interbase SYSDBA/masterkey
// // Username and password for Oracle scott/tiger
// // Username and password for JDataStore system/manager
```
/** Handle the Connect button */
void jbtConnect_actionPerformed(ActionEvent e) {
    // Get database information from the user input
    String driver = (String)jcboDriver.getSelectedItem();
    String url = (String)jcboURL.getSelectedItem();
    String username = jtfUsername.getText().trim();
    String password = new String(jpfPassword.getPassword());

    // Connection to the database
    try {
        Class.forName(driver);
        System.out.println("Driver " + driver + " loaded\n");
        connection = DriverManager.getConnection(
            url, username, password);
        System.out.println("Connected to " + url + '\n");
        setVisible(false);
    }
    catch (java.lang.Exception ex) {
        ex.printStackTrace();
    }
}

/** Handle the Cancel button */
void jbtCancel_actionPerformed(ActionEvent e) {
    // Close the dialog box
    setVisible(false);
}

/** Return connection */
public java.sql.Connection getConnection() {
    return connection;
}

Example Review

An example of using this dialog component will be given in the next example.

The modal property of the dialog box is set to true to ensure that the Connect to Database dialog is dealt with before performing other tasks.

You can choose a JDBC driver (or a database URL) from the combo box or enter a driver class (or a database URL) in the combo box.

Once a connection is established, the getConnection method returns the Connection object. If the return value is null, there is no active connection.

Example 12.5 Creating an Interactive SQL Client
This example creates a Java applet for submitting and executing SQL commands interactively, as shown in Figure 12.16. The client can connect to any JDBC data source, and submit SQL SELECT commands and non-SELECT commands for execution. The execution result is displayed for the SELECT queries, and the execution status is displayed for the non-SELECT commands.

**Figure 12.16**

You can connect to any JDBC data source and execute SQL command interactively.

Here are the steps to create this applet.

1. In the jdbcdemo project, create a new applet named SQLClient.java using the Applet wizard from the Object Gallery.

2. In the UI Designer for SQLClient.java, drop a JPanel to the north of the applet to create jPanel1, and drop a JScrollPane to the center of the applet to create jScrollPanel.

3. In jPanel1, set the layout to BorderLayout. Drop a JPanel to the north of jPanel1 to create jPanel2. Drop a JScrollPane to the center of jPanel1 to create jScrollPane2. Drop a JPanel to the south of jPanel1 to create jPanel3.

4. In jPanel2, set the layout to BorderLayout. Drop a JLabel to the west of jPanel2 and rename the label jlblConnectionStatus. Drop a JButton to the east of jPanel2 and rename the button jbtnConnect. Set the text of the button to Connect.

5. In jScrollPane2, choose titled in the border property to create titledBorder1. Set the title property of titledBorder1 to “Enter a SQL Command.”
Drop a `JTextArea` into `jScrollPane2` to create `jtaSQLCommand`.

6. In `jPanel3`, set the layout to `BorderLayout`. Drop a `JLabel` to the west of `jPanel3` and rename the label `jlblExecutionStatus`. Drop a `JButton` to the east of `jPanel3` and rename the button `jbtExecuteSQL`. Set the text of the button to “Execute SQL Command.”

7. In `jScrollPane1`, choose titled in the border property to create `titledBorder2`. Set the title property of `titledBorder2` to “SQL Execution Result.” Drop a `JTextArea` into `jScrollPane1` to create `jtaSQLResult`.

8. Install the `DBConnectDialog` component into the Component palette. `DBConnectDialog` was created in Example 12.4. Drop a `DBConnectDialog` from the Component palette to the UI Designer to create `dBConnectDialog1`.

8. Generate and implement the handlers for the Connect button and the Execute SQL Command button, as shown in Example 12.6.

Example 12.6: SQLClient.java

```java
package jdbcdemo;

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import javax.swing.*;
import javax.swing.border.*;
import java.sql.*;

public class SQLClient extends JApplet {
   // Connection to the database
   Connection connection;
   // Statement to execute SQL commands
   Statement statement;

   boolean isStandalone = false;
   JPanel jPanel1 = new JPanel();
   JPanel jPanel2 = new JPanel();
   BorderLayout borderLayout1 = new BorderLayout();
   JScrollPane jScrollPane1 = new JScrollPane();
   JTextArea jtasqlCommand = new JTextArea();
   JLabel jlblStatus = new JLabel();
   JButton jbtConnectDB = new JButton();
   BorderLayout borderLayout2 = new BorderLayout();
   TitledBorder titledBorder1;
   JScrollPane jScrollPane2 = new JScrollPane();
```
/**Initialize the applet*/
public void init() {
    try {
        jbInit();
    } catch(Exception e) {
        e.printStackTrace();
    }
}

/**Component initialization*/
private void jbInit() throws Exception {
    titledBorder1 = new TitledBorder("";
    titledBorder2 = new TitledBorder("";
    this.setSize(new Dimension(400, 300));
    jPanel1.setLayout(borderLayout1);
    jtasqlCommand.setPreferredSize(new Dimension(57, 50));
    jlblStatus.setText("No connection now");
    jbtConnectDB.setText("Connect to Database");
    jbtConnectDB.addActionListener(
        new java.awt.event.ActionListener() {
            public void actionPerformed(ActionEvent e) {
                jbtConnectDB_actionPerformed(e);
            }
        });
    jPanel2.setLayout(borderLayout2);
    jScrollPane1.setVerticalScrollBarPolicy( JScrollBar.VERTICAL_SCROLLBAR_ALWAYS);
    jScrollPane1.setBorder(titledBorder1);
    titledBorder1.setTitle("Enter a SQL Command");
    jbtExecuteSQL.setText("Execute SQL Command");
    jbtExecuteSQL.addActionListener(
        new java.awt.event.ActionListener() {
            public void actionPerformed(ActionEvent e) {
                jbtExecuteSQL_actionPerformed(e);
            }
        });
    jPanel3.setLayout(borderLayout3);
    jScrollPane2.setBorder(titledBorder2);
    titledBorder2.setTitle("SQL Execution Result");
    jlblExecutionStatus.setText("Execution status to be displayed");
    this.getContentPane().add(jPanel1, BorderLayout.NORTH);
    jPanel1.add(jPanel2, BorderLayout.NORTH);
    jPanel2.add(jlblStatus, BorderLayout.CENTER);
    jPanel2.add(jbtConnectDB, BorderLayout.EAST);
    jPanel3.add(jScrollPane1, BorderLayout.CENTER);
    jPanel3.add(jScrollPane2, BorderLayout.CENTER);
}
**Connect to DB/**

```java
void jbtConnectDB_actionPerformed(ActionEvent e) {
    dBConnectDialog1.setVisible(true);
    connection = dBConnectDialog1.getConnection();
    if (connection != null) {
        jlblStatus.setText("Connected to " + connection);
    }
}
```

**Execute SQL commands/**

```java
void jbtExecuteSQL_actionPerformed(ActionEvent e) {
    if (connection == null) {
        jtaSQLResult.setText("Please connect to a database first");
        return;
    } else {
        String sqlCommand = jtasqlCommand.getText().trim();
        if (sqlCommand.toUpperCase().startsWith("SELECT")) {
            processSQLSelect(sqlCommand);
        } else {
            processSQLNonSelect(sqlCommand);
        }
    }
}
```

**Execute SQL SELECT commands/**

```java
private void processSQLSelect(String sqlCommand) {
    try {
        // Get a new statement for the current connection
        statement = connection.createStatement();
        // Execute a SELECT SQL command
        ResultSet resultSet = statement.executeQuery(sqlCommand);
        // Find the number of columns in the result set
        int columnCount = resultSet.getMetaData().getColumnCount();
        String row = "";
        while (resultSet.next()) {
            for (int i=1; i<=columnCount; i++) {
                row += resultSet.getString(i) + " ";
            }
            jtaSQLResult.append(row + '\n');
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```
// Reset row to empty
row = "";
}

try {
    // Get a new statement for the current connection
    statement = connection.createStatement();

    // Execute a non-SELECT SQL command
    statement.executeUpdate(sqlCommand);

    jlblExecutionStatus.setText("SQL command executed");
} catch (SQLException ex) {
    // Set status label to error message
    jlblExecutionStatus.setText(ex.toString());
}

/** Execute SQL DDL, and modification commands */
private void processSQLNonSelect(String sqlCommand) {
    try {
        // Get a new statement for the current connection
        statement = connection.createStatement();

        // Execute a non-SELECT SQL command
        statement.executeUpdate(sqlCommand);

        jlblExecutionStatus.setText("SQL command executed");
    } catch (SQLException ex) {
        // Set status label to error message
        jlblExecutionStatus.setText(ex.toString());
    }

Example Review

The Connect button displays the dialog box for connecting to a database. You can retrieve the connection from the getConnection method in dBConnectDialog1.

The Execute SQL Command button gets the SQL command from the text area (jtaSQLCommand), and determines whether the command is a SELECT query or a DDL or data modification statement. If the command is a SELECT query, the executeQuery method is invoked to obtain the query result. The result is displayed in the text area (jtaSQLResult). If the command is a non-SELECT query, the executeUpdate method is invoked to execute the SQL command. The result of execution is displayed in the label (jlblExecutionStatus).

The getMetaData method in the ResultSet interface is used to obtain an instance of ResultSetMetaData. ResultSetMetaData can be used to find the metadata information on the result set, including the number of the columns in the result set. ResultSetMetaData will be further discussed in Chapter 13. In Exercise 13.3, you will modify this program to display the query result in a table.
If you connect to an Oracle database, do not use semicolon at the end of the SQL command.

**JDataStore Database**

JDataStore is an embedded relational database system developed by Borland using Java. It is bundled in JBuilder 4 and 5 Professional and Enterprise Editions. Since it is small and developed in 100% pure Java, it can be used in various types of embedded systems. It is SQL-92 compliant, and it requires no administration. The entire database is stored in one file, so it is easy to transport. You can use JDataStore databases through the JDataStore drivers locally or remotely.

**JDataStore Explorer**

JDataStore Explorer is a utility for managing and manipulating JDataStore databases. You can use it to browse and edit the contents of a JDataStore database.

To create a JDataStore database, choose Tools, JDataStore Explorer to display the JDataStore Explorer, as shown in Figure 12.17.

![JDataStore Explorer](image)

**Figure 12.17**

*JDataStore Explorer can be used to manage JDataStore databases.*

To create a new database, choose File, New to display the New JDataStore dialog box, as shown in Figure 12.18. Type C:\radjb\LiangBookDB.jds in the File name field, and choose all default options. Click OK to create the database.
Figure 12.18

You can create a JDataStore database in the New JDataStore dialog box.

You can create tables and indexes, use SQL statements, and create user accounts from the Tools menu. If you don’t create a user, you can open the database with any username without password.

You can manipulate tables in a JDataStore database using the SQL commands from the SQL window, as shown in Figure 12.19. The SQL window can be displayed by choosing Tools, SQL.
Figure 12.19

The SQL window enables you to type and execute SQL commands.

You can also browse and edit the tables by choosing the table in the left pane and choose the View tab to display the table contents in the right pane, as shown in Figure 12.20.

Figure 12.20

You can browse and edit the contents of a table in the View page in the JDataStore Explorer.

JDataStore JDBC Driver and URL

You can develop Java clients to access the JDataStore databases through the JDBC drivers. The JDataStore JDBC driver is com.borland.datastore.jdbc.DataStoreDriver. To access a local JDataStore database, use the URL


For example, the URL is jdbc:borland:dslocal:c:\radjb\LiangBookDB.jds to use the local JDataStore c:\radjb\LiangBookDB.jds.

To access a remote JDataStore database, use the URL

jdbc:borland:dsremote://<hostname>/<filename>

For example, the URL is jdbc:borland:dsremote://liang.Armstrong.edu/c:/radjb/LiangBookDB.jds to use the remote JDataStore on host liang.Armstrong.edu.

JDBC Explorer
JBuilder Professional and Enterprise has the JDBC Explorer. This is a pure-Java utility that enables you to browse a set of databases, its associated tables, views, indexes, primary keys, and stored procedures in addition to database metadata. You can also use it to edit data. To display the JDBC Explorer, choose Tools, JDBC Explorer from the JBuilder menu, as shown in Figure 12.21.

Figure 12.21

The JDBC Explorer enables you to browse databases, and execute SQL statements.

The JDBC Explorer has two panes. The left pane, referred to as the structure pane, displays the database URLs, and a hierarchical view of the tables, views, indexes, primary keys, and stored procedures. The right pane, referred to as the content pane, displays contents of the tables, views, indexes, and primary keys.

To add a new database in the JDBC Explorer, highlight the node Database URLs in structure pane, choose File, New to display the New URL dialog box for entering a driver and a URL. When a database URL is opened, you can expand the tree to display child objects. Columns are child objects of a particular database table. As in the figure above, when the Column object is selected for a table, the Summary page in the right pane lists the columns, their data type, size, and other information.

To display the contents of a table, enter a SQL select statement in the Enter SQL tab of the content pane. You will see the contents of the query displayed in the content pane after clicking the Execute button, as shown in Figure 12.22.
You can view the contents of the tables from the JDBC Explorer.

**Editing Data**

You can edit data in the JDataStore databases through the JDBC Explorer. For a demonstration, create a JDataStore database, and create a table in the database. Select the table in the navigation pane and click the Data tab in the content pane, as shown in Figure 12.23. You can now edit the data in the content pane. For more information, please refer to "JDBC Explorer User Guide" in JBuilder Help.

Figure 12.23

You can view the contents of the tables from the JDBC Explorer.
Chapter Summary

This chapter introduced JDBC, which is the Java API for accessing relational databases. You learned how to write a simple JDBC program by loading the driver, connecting to the database, creating statements, executing the statements, and processing statements. This chapter gave step-by-step instructions for working with four types of representative JDBC drivers. This chapter also provided a database connection dialog bean component, and a universal interactive SQL client.

Review Questions

12.1 Describe the following JDBC interfaces: Driver, Connection, Statement, and ResultSet.

12.2 How do you load a JDBC driver?

12.3 How do you create a JDBC connection? What is the syntax for a JDBC URL?

12.4 How do you create an instance of Statement? How do you execute a statement? Does every execution of the statement return a ResultSet?

12.5 How do you retrieve values in a ResultSet?

12.6 What are the four types of JDBC drivers? Which types of drivers require the DBMS client library? Which types of drivers can be downloaded and maintained on the server side?

Programming Exercises

12.1 Write an applet to show the number of visits made to a Web page. The count should be stored on the server side in a database table. Every time the page is visited or reloaded, the applet should send a request to the server, and the server should increase the count and send it to the applet. The applet should then display the count in a message, such as You are visitor number: 1000.

12.2 Write an applet to get stock quotes. The stock information is stored in a table consisting of six fields: stock symbol, stock share value, the high of the day, the low of the day, the highest value and lowest value in the last 52 weeks. To get a stock quote, the user enters the stock symbol and presses ENTER in the text field for the stock symbol to display the stock information.

12.3 Write a Java applet to view, insert, and update staff information stored in a database, as shown in Figure 12.24. The Staff table is created as follows:

cREATE TABLE Staff
The applet lets you view, insert, and update staff information.

12.4 Modify Example 12.5 to add a File menu with a menu item named Connect to connect a database using the DBConnectDialog component, as shown in Figure 12.25.

The Connect menu item connects to a database using the DBConnectDialog component.
12.5 Rewrite Exercise 4.8 to display the number of employees in each department in a pie chart and a bar chart, as shown in Figure 12.26. The number of employees for each department can be obtained dynamically from the database tables Department and Employee. The Department table has three columns: deptno, deptname, and deptchair. The Employee table has four columns: empno, name, office, and deptno. The number of employees in each department can be obtained using the following SQL statement:

```sql
select deptname, count(*)
from department, employee
where department.deptno = employee.deptno
group by deptname;
```

The PieChart component and the BarChart component were created in Exercise 4.8.

**Figure 12.26**

The PieChart and BarChart display the query data obtained from the data module.