CHAPTER 38

ADVANCED JAVA DATABASE PROGRAMMING

Objectives

- To create a universal SQL client for accessing local or remote database (§38.2).
- To execute SQL statements in a batch mode (§38.3).
- To process updatable and scrollable result sets (§38.4).
- To simplify Java database programming using `RowSet` (§38.5).
- To create a custom table model for `RowSet` (§38.5).
- To store and retrieve images in JDBC (§38.7).

38.1 Introduction

The preceding chapter introduced JDBC's basic features. This chapter covers its advanced features. You will learn how to develop a universal SQL client for accessing any local or remote relational database, learn how to execute statements in a batch mode to improve performance, learn scrollable result sets and how to update a database through result sets, learn how to use `RowSet` to simplify database access, and learn how to store and retrieve images.

38.2 A Universal SQL Client

In the preceding chapter, you used various drivers to connect to the database, created statements for executing SQL statements, and processed the results from SQL queries. This section presents a universal SQL client that enables you to connect to any relational database and execute SQL commands interactively, as shown in Figure 38.1. The client can connect to any JDBC data source and can 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. Listing 38.1 gives the program.

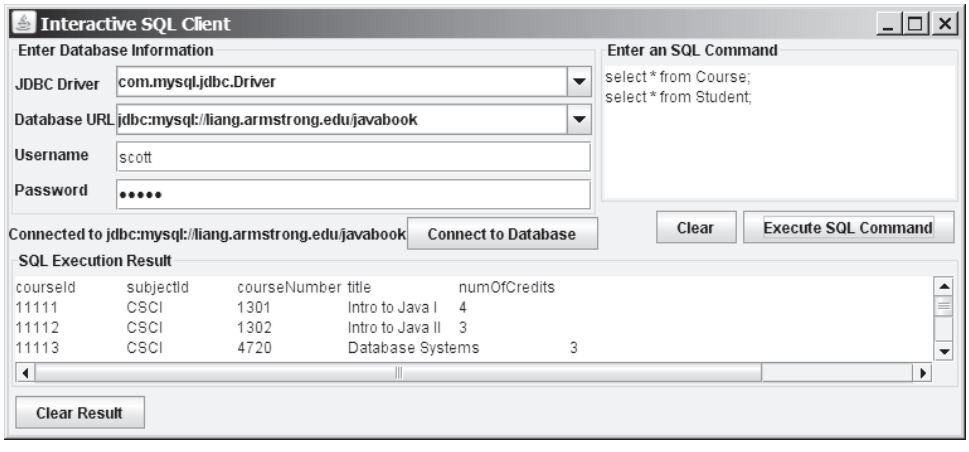


FIGURE 38.1 You can connect to any JDBC data source and execute SQL commands interactively.

LISTING 38.1 SQLClient.java

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4  import javax.swing.border.*;
5  import java.sql.*;
6  import java.util.*;
7
8  public class  SQLClient extends JApplet {
9    // Connection to the database
10   private Connection connection;
11
12   // Statement to execute SQL commands
13   private Statement statement;
14
15   // Text area to enter SQL commands
16   private JTextArea jtasqlCommand = new JTextArea();
17
```

connection

statement

```
18  // Text area to display results from SQL commands
19  private JTextArea jtaSQLResult = new JTextArea();
20
21  // JDBC info for a database connection
22  JTextField jtfUsername = new JTextField();
23  JPasswordField jpfPassword = new JPasswordField();
24  JComboBox jcboURL = new JComboBox(new String[] {                    URLs
25    "jdbc:mysql://liang.armstrong.edu/javabook",
26    "jdbc:odbc:exampleMDBDataSource",
27    "jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl"});
28  JComboBox jcboDriver = new JComboBox(new String[] {                 drivers
29    "com.mysql.jdbc.Driver", "sun.jdbc.odbc.JdbcOdbcDriver",
30    "oracle.jdbc.driver.OracleDriver"});
31
32  JButton jbtExecuteSQL = new JButton("Execute SQL Command");
33  JButton jbtClearSQLCommand = new JButton("Clear");
34  JButton jbtConnectDB1 = new JButton("Connect to Database");
35  JButton jbtClearSQLResult = new JButton("Clear Result");
36
37  // Create titled borders
38  Border titledBorder1 = new TitledBorder("Enter an SQL Command");
39  Border titledBorder2 = new TitledBorder("SQL Execution Result");
40  Border titledBorder3 = new TitledBorder(
41    "Enter Database Information");
42
43  JLabel jlblConnectionStatus = new JLabel("No connection now");
44
45  /** Initialize the applet */
46  public void init() {
47    JScrollPane jScrollPane1 = new JScrollPane(jtasqlCommand);        create UI
48    jScrollPane1.setBorder(titledBorder1);
49    JScrollPane jScrollPane2 = new JScrollPane(jtaSQLResult);
50    jScrollPane2.setBorder(titledBorder2);
51
52    JPanel jPanel1 = new JPanel(new FlowLayout(FlowLayout.RIGHT));
53    jPanel1.add(jbtClearSQLCommand);
54    jPanel1.add(jbtExecuteSQL);
55
56    JPanel jPanel2 = new JPanel();
57    jPanel2.setLayout(new BorderLayout());
58    jPanel2.add(jScrollPane1, BorderLayout.CENTER);
59    jPanel2.add(jPanel1, BorderLayout.SOUTH);
60    jPanel2.setPreferredSize(new Dimension(100, 100));
61
62    JPanel jPanel3 = new JPanel();
63    jPanel3.setLayout(new BorderLayout());
64    jPanel3.add(jlblConnectionStatus, BorderLayout.CENTER);
65    jPanel3.add(jbtConnectDB1, BorderLayout.EAST);
66
67    JPanel jPanel4 = new JPanel();
68    jPanel4.setLayout(new GridLayout(4, 1, 10, 5));
69    jPanel4.add(jcboDriver);
70    jPanel4.add(jcboURL);
71    jPanel4.add(jtfUsername);
72    jPanel4.add(jpfPassword);
73
74    JPanel jPanel5 = new JPanel();
75    jPanel5.setLayout(new GridLayout(4, 1));
76    jPanel5.add(new JLabel("JDBC Driver"));
77    jPanel5.add(new JLabel("Database URL"));
```

```
78      jPanel5.add(new JLabel("Username"));
79      jPanel5.add(new JLabel("Password"));
80
81      JPanel jPanel6 = new JPanel();
82      jPanel6.setLayout(new BorderLayout());
83      jPanel6.setBorder(titledBorder3);
84      jPanel6.add(jPanel4, BorderLayout.CENTER);
85      jPanel6.add(jPanel5, BorderLayout.WEST);
86
87      JPanel jPanel7 = new JPanel();
88      jPanel7.setLayout(new BorderLayout());
89      jPanel7.add(jPanel3, BorderLayout.SOUTH);
90      jPanel7.add(jPanel6, BorderLayout.CENTER);
91
92      JPanel jPanel8 = new JPanel();
93      jPanel8.setLayout(new BorderLayout());
94      jPanel8.add(jPanel2, BorderLayout.CENTER);
95      jPanel8.add(jPanel7, BorderLayout.WEST);
96
97      JPanel jPanel9 = new JPanel(new FlowLayout(FlowLayout.LEFT));
98      jPanel9.add(jbtClearSQLResult);
99
100     jcboURL.setEditable(true);
101     jcboDriver.setEditable(true);
102
103     add(jPanel8, BorderLayout.NORTH);
104     add(jScrollPane2, BorderLayout.CENTER);
105     add(jPanel9, BorderLayout.SOUTH);
106
107     jbtExecuteSQL.addActionListener(new ActionListener() {
108       public void actionPerformed(ActionEvent e) {
109         executeSQL();
110       }
111     });
112     jbtConnectDB1.addActionListener(new ActionListener() {
113       public void actionPerformed(ActionEvent e) {
114         connectToDB();
115       }
116     });
117     jbtClearSQLCommand.addActionListener(new ActionListener() {
118       public void actionPerformed(ActionEvent e) {
119         jtasqlCommand.setText(null);
120       }
121     });
122     jbtClearSQLResult.addActionListener(new ActionListener() {
123       public void actionPerformed(ActionEvent e) {
124         jtaSQLResult.setText(null);
125       }
126     });
127   }
128
129   /** Connect to DB */
130   private void connectToDB() {
131     // Get database information from the user input
132     String driver = (String)jcboDriver.getSelectedItem();
133     String url = (String)jcboURL.getSelectedItem();
134     String username = jtfUsername.getText().trim();
135     String password = new String(jpfPassword.getPassword());
136
```

execute SQL

connect to database

clear command

clear result

```
137     // Connection to the database
138     try {
139       Class.forName(driver);                                    load driver
140       connection = DriverManager.getConnection(                 connect database
141         url, username, password);
142       jlblConnectionStatus.setText("Connected to " + url);
143     }
144     catch (java.lang.Exception ex) {
145       ex.printStackTrace();
146     }
147   }
148
149   /** Execute SQL commands */
150   private void executeSQL() {
151     if (connection == null) {
152       jtaSQLResult.setText("Please connect to a database first");
153       return;
154     }
155     else {
156       String sqlCommands = jtasqlCommand.getText().trim();
157       String[] commands = sqlCommands.replace('\n', ' ').split(";");
158
159       for (String aCommand: commands) {
160         if (aCommand.trim().toUpperCase().startsWith("SELECT")) {
161           processSQLSelect(aCommand);                           process SQL select
162         }
163         else {
164           processSQLNonSelect(aCommand);                        process non-select
165         }
166       }
167     }
168   }
169
170   /** Execute SQL SELECT commands */
171   private void processSQLSelect(String sqlCommand) {
172     try {
173       // Get a new statement for the current connection
174       statement = connection.createStatement();
175
176       // Execute a SELECT SQL command
177       ResultSet resultSet = statement.executeQuery(sqlCommand);
178
179       // Find the number of columns in the result set
180       int columnCount = resultSet.getMetaData().getColumnCount();
181       String row = "";
182
183       // Display column names
184       for (int i = 1; i <= columnCount; i++) {
185         row += resultSet.getMetaData().getColumnName(i) + "\t";
186       }
187
188       jtaSQLResult.append(row + '\n');
189
190       while (resultSet.next()) {
191         // Reset row to empty
192         row = "";
193
194         for (int i = 1; i <= columnCount; i++) {
195           // A non-String column is converted to a string
```

```
196           row += resultSet.getString(i) + "\t";
197         }
198
199         jtaSQLResult.append(row + '\n');
200       }
201     }
202     catch (SQLException ex) {
203       jtaSQLResult.setText(ex.toString());
204     }
205   }
206
207   /** Execute SQL DDL, and modification commands */
208   private void processSQLNonSelect(String sqlCommand) {
209     try {
210       // Get a new statement for the current connection
211       statement = connection.createStatement();
212
213       // Execute a non-SELECT SQL command
214       statement.executeUpdate(sqlCommand);
215
216       jtaSQLResult.setText("SQL command executed");
217     }
218     catch (SQLException ex) {
219       jtaSQLResult.setText(ex.toString());
220     }
221   }
222 }
```

main method omitted

The user selects or enters the JDBC driver, database URL, username, and password, and clicks the *Connect to Database* button to connect to the specified database using the `connectToDB()` method (lines 130–147).

When the user clicks the *Execute SQL Command* button, the `executeSQL()` method is invoked (lines 150–168) to get the SQL commands from the text area (`jtaSQLCommand`) and extract each command separated by a semicolon (;). It then determines whether the command is a SELECT query or a DDL or data modification statement (lines 160–165). If the command is a SELECT query, the `processSQLSelect` method is invoked (lines 171–205). This method uses the `executeQuery` method (line 177) to obtain the query result. The result is displayed in the text area (`jtaSQLResult`). If the command is a non-SELECT query, the `processSQLNonSelect()` method is invoked (lines 208–221). This method uses the `executeUpdate` method (line 214) to execute the SQL command.

The `getMetaData` method (lines 180, 185) in the `ResultSet` interface is used to obtain an instance of `ResultSetMetaData`. The `getColumnCount` method (line 180) returns the number of columns in the result set, and the `getColumnName(i)` method (line 185) returns the column name for the *i*th column.

38.3 Batch Processing

In all the preceding examples, SQL commands are submitted to the database for execution one at a time. This is inefficient for processing a large number of updates. For example, suppose you wanted to insert a thousand rows into a table. Submitting one INSERT command at a time would take nearly a thousand times longer than submitting all the INSERT commands in a batch at once. To improve performance, JDBC introduced the batch update for processing nonselect SQL commands. A batch update consists of a sequence of nonselect SQL commands. These commands are collected in a batch and submitted to the database all together.

To use the batch update, you add nonselect commands to a batch using the `addBatch` method in the `Statement` interface. After all the SQL commands are added to the batch, use the `executeBatch` method to submit the batch to the database for execution.

For example, the following code adds a create table command, adds two insert statements in a batch, and executes the batch.

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

// Add SQL commands to the batch
statement.addBatch("create table T (C1 integer, C2 varchar(15))");
statement.addBatch("insert into T values (100, 'Smith')");
statement.addBatch("insert into T values (200, 'Jones')");

// Execute the batch
int count[] = statement.executeBatch();
```

The `executeBatch()` method returns an array of counts, each of which counts the number of rows affected by the SQL command. The first count returns `0` because it is a DDL command. The other counts return `1` because only one row is affected.

Note

To find out whether a driver supports batch updates, invoke `supportsBatchUpdates()` on a `DatabaseMetaData` instance. If the driver supports batch updates, it will return true. The JDBC drivers for MySQL, Access, and Oracle all support batch updates.

To demonstrate batch processing, consider writing a program that gets data from a text file and copies the data from the text file to a table, as shown in Figure 38.2. The text file consists of lines that each corresponds to a row in the table. The fields in a row are separated by commas. The string values in a row are enclosed in single quotes. You can view the text file by clicking the *View File* button and copy the text to the table by clicking the *Copy* button. The table must already be defined in the database. Figure 38.2 shows the text file table.txt copied to table Person. Person is created using the following statement:

```
create table Person (
  firstName varchar(20),
  mi char(1),
  lastName varchar(20)
)
```

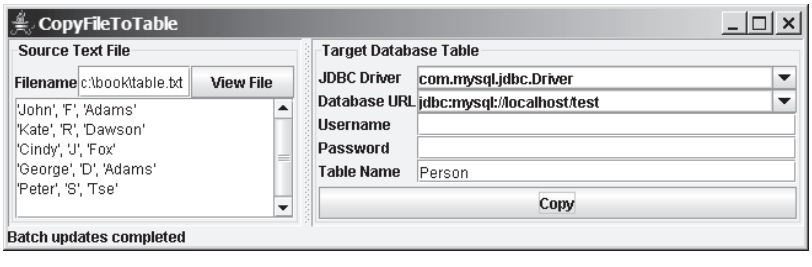


FIGURE 38.2 The `CopyFileToTable` utility copies text files to database tables.

Listing 38.2 gives the solution to the problem.

LISTING 38.2 CopyFileToTable.java

```
1 import javax.swing.*;
2 import javax.swing.border.*;
3 import java.awt.*;
```

```
 4 import java.awt.event.*;
 5 import java.io.*;
 6 import java.sql.*;
 7 import java.util.*;
 8
 9 public class  CopyFileToTable extends JFrame {
10   // Text file info
11   private JTextField jtfFilename = new JTextField();
12   private JTextArea jtaFile = new JTextArea();
13
14   // JDBC and table info
15   private JComboBox jcboDriver = new JComboBox(new String[] {
16     "com.mysql.jdbc.Driver", "sun.jdbc.odbc.JdbcOdbcDriver",
17     "oracle.jdbc.driver.OracleDriver"});
18   private JComboBox jcboURL = new JComboBox(new String[] {
19     "jdbc:mysql://localhost/javabook",
20     "jdbc:odbc:exampleMDBDataSource",
21     "jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl"});
22   private JTextField jtfUsername = new JTextField();
23   private JPasswordField jtfPassword = new JPasswordField();
24   private JTextField jtfTableName = new JTextField();
25
26   private JButton jbtViewFile = new JButton("View File");
27   private JButton jbtCopy = new JButton("Copy");
28   private JLabel jlblStatus = new JLabel();
29
30   public CopyFileToTable() {
31     JPanel jPane1 = new JPanel();
32     jPane1.setLayout(new BorderLayout());
33     jPane1.add(new JLabel("Filename"), BorderLayout.WEST);
34     jPane1.add(jbtViewFile, BorderLayout.EAST);
35     jPane1.add(jtfFilename, BorderLayout.CENTER);
36
37     JPanel jPane2 = new JPanel();
38     jPane2.setLayout(new BorderLayout());
39     jPane2.setBorder(new TitledBorder("Source Text File"));
40     jPane2.add(jPane1, BorderLayout.NORTH);
41     jPane2.add(new JScrollPane(jtaFile), BorderLayout.CENTER);
42
43     JPanel jPane3 = new JPanel();
44     jPane3.setLayout(new GridLayout(5, 0));
45     jPane3.add(new JLabel("JDBC Driver"));
46     jPane3.add(new JLabel("Database URL"));
47     jPane3.add(new JLabel("Username"));
48     jPane3.add(new JLabel("Password"));
49     jPane3.add(new JLabel("Table Name"));
50
51     JPanel jPane4 = new JPanel();
52     jPane4.setLayout(new GridLayout(5, 0));
53     jcboDriver.setEditable(true);
54     jPane4.add(jcboDriver);
55     jcboURL.setEditable(true);
56     jPane4.add(jcboURL);
57     jPane4.add(jtfUsername);
58     jPane4.add(jtfPassword);
59     jPane4.add(jtfTableName);
60
61     JPanel jPane5 = new JPanel();
62     jPane5.setLayout(new BorderLayout());
63     jPane5.setBorder(new TitledBorder("Target Database Table"));
```

drivers

URLs

create UI

```
64      jPane5.add(jbtCopy, BorderLayout.SOUTH);
65      jPane5.add(jPane3, BorderLayout.WEST);
66      jPane5.add(jPane4, BorderLayout.CENTER);
67
68      add(jlblStatus, BorderLayout.SOUTH);
69      add(new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
70        jPane2, jPane5), BorderLayout.CENTER);
71
72      jbtViewFile.addActionListener(new ActionListener() {
73        public void actionPerformed(ActionEvent evt) {
74          showFile();                                              view file
75        }
76      });
77
78      jbtCopy.addActionListener(new ActionListener() {
79        public void actionPerformed(ActionEvent evt) {
80          try {
81            copyFile();                                            to table
82          }
83          catch (Exception ex) {
84            jlblStatus.setText(ex.toString());
85          }
86        }
87      });
88    }
89
90    /** Display the file in the text area */
91    private void showFile() {
92      Scanner input = null;
93      try {
94        // Use a Scanner to read text from the file
95        input = new Scanner(new File(jtfFilename.getText().trim()));
96
97        // Read a line and append the line to the text area
98        while (input.hasNext())
99          jtaFile.append(input.nextLine() + '\n');
100     }
101     catch (FileNotFoundException ex) {
102       System.out.println("File not found: " + jtfFilename.getText());
103     }
104     catch (IOException ex) {
105       ex.printStackTrace();
106     }
107     finally {
108       if (input != null) input.close();
109     }
110   }
111
112   private void copyFile() throws Exception {
113     // Load the JDBC driver
114     Class.forName(((String)jcboDriver.getSelectedItem()).trim());      load driver
115     System.out.println("Driver loaded");
116
117     // Establish a connection
118     Connection conn = DriverManager.getConnection                      connect database
119       (((String)jcboURL.getSelectedItem()).trim(),
120       jtfUsername.getText().trim(),
121       String.valueOf(jtfPassword.getPassword()).trim());
122     System.out.println("Database connected");
123
```

```
124     // Read each line from the text file and insert it to the table
125     insertRows(conn);
126   }
127
128   private void insertRows(Connection connection) {
129     // Build the SQL INSERT statement
130     String sqlInsert = "insert into " + jtfTableName.getText()
131       + " values (";
132
133     // Use a Scanner to read text from the file
134     Scanner input = null;
135
136     // Get file name from the text field
137     String filename = jtfFilename.getText().trim();
138
139     try {
140       // Create a scanner
141       input = new Scanner(new File(filename));
142
143       // Create a statement
144       Statement statement = connection.createStatement();
145
146       System.out.println("Driver major version? " +
147         connection.getMetaData().getDriverMajorVersion());
148
149       // Determine if batchUpdatesSupported is supported
150       boolean batchUpdatesSupported = false;
151
152       try {
153         if (connection.getMetaData().supportsBatchUpdates()) {
154           batchUpdatesSupported = true;
155           System.out.println("batch updates supported");
156         }
157         else {
158           System.out.println("The driver " +
159             "does not support batch updates");
160         }
161       }
162       catch (UnsupportedOperationException ex) {
163         System.out.println("The operation is not supported");
164       }
165
166       // Determine if the driver is capable of batch updates
167       if (batchUpdatesSupported) {
168         // Read a line and add the insert table command to the batch
169         while (input.hasNext()) {
170           statement.addBatch(sqlInsert + input.nextLine() + ")");
171         }
172
173         statement.executeBatch();
174
175         jlblStatus.setText("Batch updates completed");
176       }
177       else {
178         // Read a line and execute insert table command
179         while (input.hasNext()) {
180           statement.executeUpdate(sqlInsert + input.nextLine() + ")");
181         }
182
183         jlblStatus.setText("Single row update completed");
```

insert row

statement

batch

execute batch

```
184       }
185     }
186     catch (SQLException ex) {
187       System.out.println(ex);
188     }
189     catch (FileNotFoundException ex) {
190       System.out.println("File not found: " + filename);
191     }
192     catch (IOException ex) {
193       ex.printStackTrace();
194     }
195     finally {
196       if (input != null) input.close();
197     }
198   }                                                    main method omitted
199 }
```

The `insertRows` method (lines 128–198) uses the batch updates to submit SQL INSERT commands to the database for execution, if the driver supports batch updates. Lines 152–164 check whether the driver supports batch updates. If the driver does not support the operation, an `UnsupportedOperationException` exception will be thrown (line 162) when the `supportsBatchUpdates()` method is invoked.

The tables must already be created in the database. The file format and contents must match the database table specification. Otherwise, the SQL INSERT command will fail.

In Exercise 38.1, you will write a program to insert a thousand records to a database and compare the performance with and without batch updates.

38.4 Scrollable and Updatable Result Set

The result sets used in the preceding examples are read sequentially. A result set maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The `next()` method moves the cursor forward to the next row. This is known as *sequential forward reading*. It is the only way of processing the rows in a result set that is supported by JDBC 1.

sequential forward reading

With the new versions of JDBC, you can scroll the rows both forward and backward and move the cursor to a desired location using the `first`, `last`, `next`, `previous`, `absolute`, or `relative` method. Additionally, you can insert, delete, or update a row in the result set and have the changes automatically reflected in the database.

To obtain a scrollable or updatable result set, you must first create a statement with an appropriate type and concurrency mode. For a static statement, use

```
Statement statement = connection.createStatement
  (int resultSetType, int resultSetConcurrency);
```

For a prepared statement, use

```
PreparedStatement statement = connection.prepareStatement
  (String sql, int resultSetType, int resultSetConcurrency);
```

The possible values of `resultSetType` are the constants defined in the `ResultSet`:

scrollable?

- `TYPE_FORWARD_ONLY`: The result set is accessed forward sequentially.
- `TYPE_SCROLL_INSENSITIVE`: The result set is scrollable, but not sensitive to changes in the database.
- `TYPE_SCROLL_SENSITIVE`: The result set is scrollable and sensitive to changes made by others. Use this type if you want the result set to be scrollable and updatable.

updatable?

The possible values of `resultSetConcurrency` are the constants defined in the `ResultSet`:

- `CONCUR_READ_ONLY`: The result set cannot be used to update the database.
- `CONCUR_UPDATABLE`: The result set can be used to update the database.

For example, if you want the result set to be scrollable and updatable, you can create a statement, as follows:

```
Statement statement = connection.createStatement
  (ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE)
```

You use the `executeQuery` method in a `Statement` object to execute an SQL query that returns a result set as follows:

```
ResultSet resultSet = statement.executeQuery(query);
```

`ResultSet` methods

You can now use the methods `first()`, `next()`, `previous()`, and `last()` to move the cursor to the first row, next row, previous row, and last row. The `absolute(int row)` method moves the cursor to the specified row; and the `getXxx(int columnIndex)` or `getXxx(String columnName)` method is used to retrieve the value of a specified field at the current row. The methods `insertRow()`, `deleteRow()`, and `updateRow()` can also be used to insert, delete, and update the current row. Before applying `insertRow` or `updateRow`, you need to use the method `updateXxx(int columnIndex, Xxx value)` or `update(String columnName, Xxx value)` to write a new value to the field at the current row. The `cancelRowUpdates()` method cancels the updates made to a row. The `close()` method closes the result set and releases its resource. The `wasNull()` method returns true if the last column read had a value of SQL NULL.

Listing 38.3 gives an example that demonstrates how to create a scrollable and updatable result set. The program creates a result set for the `StateCapital` table. The `StateCapital` table is defined as follows:

```
create table StateCapital (
  state varchar(40),
  capital varchar(40)
);
```

LISTING 38.3 ScrollUpdateResultSet.java

load driver

connect to DB

set auto commit

scrollable updatable

```
1  import java.sql.*;
2
3  public class  ScrollUpdateResultSet {
4    public static void main(String[] args)
5        throws SQLException, ClassNotFoundException {
6      // Load the JDBC driver
7      Class.forName("oracle.jdbc.driver.OracleDriver");
8      System.out.println("Driver loaded");
9
10     // Establish a connection
11     Connection connection = DriverManager.getConnection
12       ("jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl",
13        "scott", "tiger");
14     connection.setAutoCommit(true);
15     System.out.println("Database connected");
16
17     // Get a new statement for the current connection
18     Statement statement = connection.createStatement(
```

```
19        ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
20
21      // Get ResultSet
22      ResultSet resultSet = statement.executeQuery                    get result set
23        ("select state, capital from StateCapital");
24
25      System.out.println("Before update ");
26      displayResultSet(resultSet);
27
28      // Update the second row
29      resultSet.absolute(2); // Move cursor to the second row          move cursor
30      resultSet.updateString("state", "New S"); // Update the column
31      resultSet.updateString("capital", "New C"); // Update the column
32      resultSet.updateRow(); // Update the row in the data source      update row
33
34      // Insert after the last row
35      resultSet.last();
36      resultSet.moveToInsertRow(); // Move cursor to the insert row    move cursor
37      resultSet.updateString("state", "Florida");
38      resultSet.updateString("capital", "Tallahassee");
39      resultSet.insertRow(); // Insert the row                        insert row
40      resultSet.moveToCurrentRow(); // Move the cursor to the current row
41
42      // Delete fourth row
43      resultSet.absolute(4); // Move cursor to the 5th row           move cursor
44      resultSet.deleteRow(); // Delete the second row                delete row
45
46      System.out.println("After update ");
47      resultSet = statement.executeQuery
48        ("select state, capital from StateCapital");
49      displayResultSet(resultSet);
50
51      // Close the connection
52      resultSet.close();                                              close result set
53    }
54
55    private static void displayResultSet(ResultSet resultSet)        display result set
56        throws SQLException {
57      ResultSetMetaData rsMetaData = resultSet.getMetaData();
58      resultSet.beforeFirst();
59      while (resultSet.next()) {
60        for (int i = 1; i <= rsMetaData.getColumnCount(); i++)
61          System.out.printf("%-12s\t", resultSet.getObject(i));
62        System.out.println();
63      }
64    }
65  }
```

```
Driver loaded
Database connected

Before update
Indiana          Indianapolis
Illinois         Springfield
California       Sacramento
Georgia          Atlanta
Texas            Austin
```

```
After update
Indiana          Indianapolis
New S            New C
California       Sacramento
Texas            Austin
Florida          Tallahassee
```

scrollable and updatable

The code in lines 18–19 creates a `Statement` for producing scrollable and updatable result sets.

update row

The program moves the cursor to the second row in the result set (line 29), updates two columns in this row (lines 30–31), and invokes the `updateRow()` method to update the row in the underlying database (line 32).

An updatable `ResultSet` object has a special row associated with it that serves as a staging area for building a row to be inserted. This special row is called the *insert row*. To insert a row, first invoke the `moveToInsertRow()` method to move the cursor to the insert row (line 36), then update the columns using the `updateXxx` method (lines 37–38), and finally insert the row using the `insertRow()` method (line 39). Invoking `moveToCurrentRow()` moves the cursor to the current inserted row (lines 40).

insert row

insert row

The program moves to the fourth row and invokes the `deleteRow()` method to delete the row from the database (lines 43–44).

Note

driver support

Not all current drivers support scrollable and updatable result sets. The example is tested using Oracle ojdbc6 driver. You can use `supportsResultSetType(int type)` and `supportsResultSetConcurrency(int type, int concurrency)` in the `DatabaseMetaData` interface to find out which result type and currency modes are supported by the JDBC driver. But even if a driver supports the scrollable and updatable result set, a result set for a complex query might not be able to perform an update. For example, the result set for a query that involves several tables is likely not to support update operations.

Note

driver support

The program may not work, if lines 22–23 are replaced by

```
ResultSet resultSet = statement.executeQuery
  ("select * from StateCapital");
```

38.5 `RowSet`, `JdbcRowSet`, and `CachedRowSet`

JDBC introduced a new `RowSet` interface that can be used to simplify database programming. The `RowSet` interface extends `java.sql.ResultSet` with additional capabilities that allow a `RowSet` instance to be configured to connect to a JDBC url, username, and password, set an SQL command, execute the command, and retrieve the execution result. In essence, it combines `Connection`, `Statement`, and `ResultSet` into one interface.

extends `ResultSet`

Note

supported?

Not all JDBC drivers support `RowSet`. Currently, the JDBC-ODBC driver does not support all features of `RowSet`.

38.5.1 RowSet Basics

connected vs. disconnected

There are two types of `RowSet` objects: connected and disconnected. A *connected* `RowSet` object makes a connection with a data source and maintains that connection throughout its life cycle. A disconnected `RowSet` object makes a connection with a data source, executes a query to get data from the data source, and then closes the connection. A *disconnected* rowset may make changes to its data while it is disconnected and then send the changes back to the original source of the data, but it must reestablish a connection to do so.

There are several versions of `RowSet`. Two frequently used are `JdbcRowSet` and `CachedRowSet`. Both are subinterfaces of `RowSet`. `JdbcRowSet` is connected, while `CachedRowSet` is disconnected. Also, `JdbcRowSet` is neither serializable nor cloneable, while `CachedRowSet` is both. The database vendors are free to provide concrete implementations for these interfaces. Sun has provided the reference implementation `JdbcRowSetImpl` for `JdbcRowSet` and `CachedRowSetImpl` for `CachedRowSet`. Figure 38.3 shows the relationship of these components.

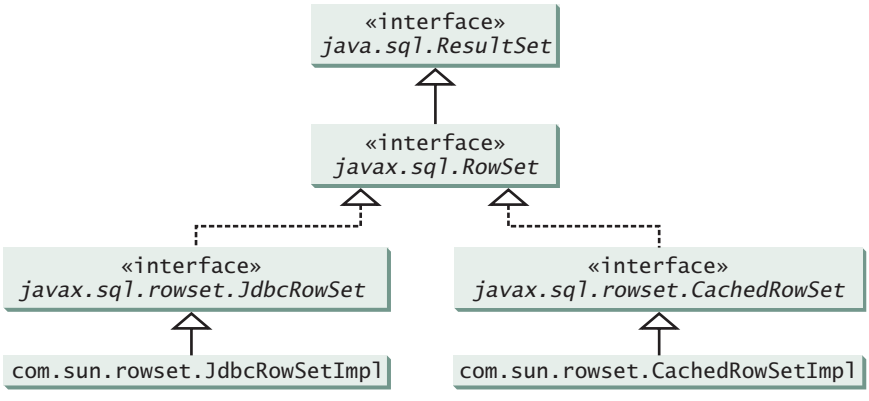


FIGURE 38.3 The `JdbcRowSetImpl` and `CachedRowSetImpl` are concrete implementations of `RowSet`.

The `RowSet` interface contains the JavaBeans properties with get and set methods. You can use the set methods to set a new url, username, password, and command for an SQL statement. Using a `RowSet`, Listing 37.1 can be simplified, as shown in Listing 38.4.

LISTING 38.4 SimpleRowSet.java

```
1  import java.sql.SQLException;
2  import javax.sql.RowSet;
3  import com.sun.rowset.*;
4
5  public class  SimpleRowSet {
6    public static void main(String[] args)
7        throws SQLException, ClassNotFoundException {
8      // Load the JDBC driver
9      Class.forName("com.mysql.jdbc.Driver");                    load driver
10     System.out.println("Driver loaded");
11
12     // Create a row set
13     RowSet rowSet = new JdbcRowSetImpl();                      create RowSet
14
15     // Set RowSet properties
16     rowSet.setUrl("jdbc:mysql://localhost/javabook");          set url
17     rowSet.setUsername("scott");                               set username
```

set password
set command
execute command
get result
close connection

```
18      rowSet.setPassword("tiger");
19      rowSet.setCommand("select firstName, mi, lastName " +
20        "from Student where lastName = 'Smith'");
21      rowSet.execute();
22
23      // Iterate through the result and print the student names
24      while (rowSet.next())
25        System.out.println(rowSet.getString(1) + "\t" +
26          rowSet.getString(2) + "\t" + rowSet.getString(3));
27
28      // Close the connection
29      rowSet.close();
30    }
31  }
```

Line 13 creates a `RowSet` object using `JdbcRowSetImpl`. The program uses the `RowSet`'s set method to set a URL, username, and password (lines 16–18) and a command for a query statement (line 19). Line 24 executes the command in the `RowSet`. The methods `next()` and `getString(int)` for processing the query result (lines 25–26) are inherited from `ResultSet`.

using **CachedRowSet**

If you replace `JdbcRowSet` with `CachedRowSet` in line 13, the program will work just fine. Note that the JDBC-ODBC driver support `JdbcRowSetImpl`, but not `CachedRowSetImpl`.

Tip

obtain metadata

Since `RowSet` is a subinterface of `ResultSet`, all the methods in `ResultSet` can be used in `RowSet`. For example, you can obtain `ResultSetMetaData` from a `RowSet` using the `getMetaData()` method.

38.5.2 `RowSet` for `PreparedStatement`

The discussion in §37.5, "PreparedStatement," introduced processing parameterized SQL statements using the `PreparedStatement` interface. `RowSet` has the capability to support parameterized SQL statements. The set methods for setting parameter values in `PreparedStatement` are implemented in `RowSet`. You can use these methods to set parameter values for a parameterized SQL command. Listing 38.5 demonstrates how to use a parameterized statement in `RowSet`. Line 19 sets an SQL query statement with two parameters for `lastName` and `mi` in a `RowSet`. Since these two parameters are strings, the `setString` method is used to set actual values in lines 21–22.

LISTING 38.5 RowSetPreparedStatement.java

load driver
create **RowSet**
set url

```
1  import java.sql.*;
2  import javax.sql.RowSet;
3  import com.sun.rowset.*;
4
5  public class  RowSetPreparedStatement {
6    public static void main(String[] args)
7        throws SQLException, ClassNotFoundException {
8      // Load the JDBC driver
9      Class.forName("com.mysql.jdbc.Driver");
10     System.out.println("Driver loaded");
11
12     // Create a row set
13     RowSet rowSet = new JdbcRowSetImpl();
14
15     // Set RowSet properties
16     rowSet.setUrl("jdbc:mysql://localhost/javabook");
17     rowSet.setUsername("scott");
```

```
18      rowSet.setPassword("tiger");
19      rowSet.setCommand("select * from Student where lastName = ? " +     SQL with parameters
20        "and mi = ?");
21      rowSet.setString(1, "Smith");                                       set parameter
22      rowSet.setString(2, "R");                                           set parameter
23      rowSet.execute();                                                   execute
24
25      ResultSetMetaData rsMetaData = rowSet.getMetaData();                metadata
26      for (int i = 1; i <= rsMetaData.getColumnCount(); i++)
27        System.out.printf("%-12s\t", rsMetaData.getColumnName(i));
28      System.out.println();
29
30      // Iterate through the result and print the student names
31      while (rowSet.next()) {
32        for (int i = 1; i <= rsMetaData.getColumnCount(); i++)
33          System.out.printf("%-12s\t", rowSet.getObject(i));
34        System.out.println();
35      }
36
37      // Close the connection
38      rowSet.close();                                                     close connection
39    }
40  }
```

38.5.3 Scrolling and Updating RowSet

By default, a `ResultSet` object is neither scrollable nor updatable. However, a `RowSet` object is both. It is easier to scroll and update a database through a `RowSet` than through a `ResultSet`. Listing 38.6 rewrites Listing 38.3 using a `RowSet`. You can use methods such as `absolute(int)` to move the cursor and methods such as `delete()`, `updateRow()`, and `insertRow()` to update the database.

LISTING 38.6 ScrollUpdateRowSet.java

```
1  import java.sql.*;
2  import javax.sql.RowSet;
3  import com.sun.rowset.JdbcRowSetImpl;
4
5  public class  ScrollUpdateRowSet {
6    public static void main(String[] args)
7        throws SQLException, ClassNotFoundException {
8      // Load the JDBC driver
9      Class.forName("oracle.jdbc.driver.OracleDriver");                   load driver
10     System.out.println("Driver loaded");
11
12     // Create a row set
13     RowSet rowSet = new JdbcRowSetImpl();                               create a RowSet
14
15     // Set RowSet properties
16     rowSet.setUrl("jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl");   set url
17     rowSet.setUsername("scott");                                        set username
18     rowSet.setPassword("tiger");                                        set password
19     rowSet.setCommand("select state, capital from StateCapital");       set SQL command
20     rowSet.execute();                                                   execute
21
22     System.out.println("Before update ");
23     displayRowSet(rowSet);                                              display rowSet
24
```

move cursor

update row

prepare insert

insert row

delete row

close **rowSet**

```
25     // Update the second row
26     rowSet.absolute(2); // Move cursor to the 2nd row
27     rowSet.updateString("state", "New S"); // Update the column
28     rowSet.updateString("capital", "New C"); // Update the column
29     rowSet.updateRow(); // Update the row in the data source
30
31     // Insert after the second row
32     rowSet.last();
33     rowSet.moveToInsertRow(); // Move cursor to the insert row
34     rowSet.updateString("state", "Florida");
35     rowSet.updateString("capital", "Tallahassee");
36     rowSet.insertRow(); // Insert the row
37     rowSet.moveToCurrentRow(); // Move the cursor to the current row
38
39     // Delete fourth row
40     rowSet.absolute(4); // Move cursor to the fifth row
41     rowSet.deleteRow(); // Delete the second row
42
43     System.out.println("After update ");
44     displayRowSet(rowSet);
45
46     // Close the connection
47     rowSet.close();
48   }
49
50   private static void displayRowSet(RowSet rowSet)
51       throws SQLException {
52     ResultSetMetaData rsMetaData = rowSet.getMetaData();
53     rowSet.beforeFirst();
54     while (rowSet.next()) {
55       for (int i = 1; i <= rsMetaData.getColumnCount(); i++)
56         System.out.printf("%-12s\t", rowSet.getObject(i));
57       System.out.println();
58     }
59   }
60 }
```

updating **CachedRowSet**

If you replace `JdbcRowSet` with `CachedRowSet` in line 13, the database is not changed. To make the changes on the `CachedRowSet` effective in the database, you must invoke the `acceptChanges()` method after you make all the changes, as follows:

```
// Write changes back to the database
((com.sun.rowset.CachedRowSetImpl)rowSet).acceptChanges();
```

This method automatically reconnects to the database and writes all the changes back to the database.

38.5.4 RowSetEvent

A `RowSet` object fires a `RowSetEvent` whenever the object's cursor has moved, a row has changed, or the entire row set has changed. This event can be used to synchronize a `RowSet` with the components that rely on the `RowSet`. For example, a visual component that displays the contents of a `RowSet` should be synchronized with the `RowSet`. The `RowSetEvent` can be used to achieve synchronization. The handlers in `RowSetListener` are `cursorMoved(RowSetEvent)`, `rowChanged(RowSetEvent)`, and `cursorSetChanged(RowSetEvent)`.

Listing 38.7 gives an example that demonstrates `RowSetEvent`. A listener for `RowSetEvent` is registered in lines 14–26. When `rowSet.execute()` (line 33) is executed, the entire

row set is changed, so the listener's `rowSetChanged` handler is invoked. When `rowSet.last()` (line 35) is executed, the cursor is moved, so the listener's `cursorMoved` handler is invoked. When `rowSet.updateRow()` (line 37) is executed, the row is updated, so the listener's `rowChanged` handler is invoked.

LISTING 38.7 TestRowSetEvent.java

```
1  import java.sql.*;
2  import javax.sql.*;
3  import com.sun.rowset.*;
4
5  public class TestRowSetEvent {
6    public static void main(String[] args)
7        throws SQLException, ClassNotFoundException {
8      // Load the JDBC driver
9      Class.forName("com.mysql.jdbc.Driver");                    load driver
10     System.out.println("Driver loaded");
11
12     // Create a row set
13     RowSet rowSet = new CachedRowSetImpl();                    create RowSet
14     rowSet.addRowSetListener(new RowSetListener() {            register listener
15       public void cursorMoved(RowSetEvent e) {
16         System.out.println("Cursor moved");
17       }
18
19       public void rowChanged(RowSetEvent e) {
20         System.out.println("Row changed");
21       }
22
23       public void rowSetChanged(RowSetEvent e) {
24         System.out.println("row set changed");
25       }
26     });
27
28     // Set RowSet properties
29     rowSet.setUrl("jdbc:mysql://localhost/javabook");
30     rowSet.setUsername("scott");
31     rowSet.setPassword("tiger");
32     rowSet.setCommand("select * from Student");
33     rowSet.execute();                                          row set changed
34
35     rowSet.last(); // Cursor moved                             cursor moved
36     rowSet.updateString("lastName", "Yao"); // Update column
37     rowSet.updateRow(); // Row updated                         row updated
38
39     // Close the connection
40     rowSet.close();
41   }
42 }
```

38.6 Custom RowSetTableModel

Often you need to display a query result set in a `JTable`. You may define a table model for a row set and plug this model to a `JTable`. To define a table model, extend the `AbstractTableModel` class and implement at least three methods: `getRowCount()`, `getColumnCount()`, and `getValueAt(int row, int column)`. The `AbstractTableModel` class was introduced in §36.3, "Table Models and Table Column Models."

Listing 38.8 shows the `RowSetTableModel` class.

LISTING 38.8 RowSetTableModel.java

```
1  import java.sql.*;
2  import javax.sql.*;
3  import javax.swing.table.AbstractTableModel;
4
5  public class RowSetTableModel extends AbstractTableModel
6      implements RowSetListener {
7    // RowSet for the result set
8    private RowSet rowSet;
9
10   /** Return the rowset */
11   public RowSet getRowSet() {
12     return rowSet;
13   }
14
15   /** Set a new rowset */
16   public void setRowSet(RowSet rowSet) {
17     if (rowSet != null) {
18       this.rowSet = rowSet;
19       rowSet.addRowSetListener(this);
20       fireTableStructureChanged();
21     }
22   }
23
24   /** Return the number of rows in the row set */
25   public int getRowCount() {
26     try {
27       rowSet.last();
28       return rowSet.getRow();
29     }
30     catch (Exception ex) {
31       ex.printStackTrace();
32     }
33
34     return 0;
35   }
36
37   /** Return the number of columns in the row set */
38   public int getColumnCount() {
39     try {
40       if (rowSet != null) {
41         return rowSet.getMetaData().getColumnCount();
42       }
43     }
44     catch (SQLException ex) {
45       ex.printStackTrace();
46     }
47
48     return 0;
49   }
50
51   /** Return value at the specified row and column */
52   public Object getValueAt(int row, int column) {
53     try {
54       rowSet.absolute(row + 1);
55       return rowSet.getObject(column + 1);
56     }
```

rowSet

getRowSet

setRowSet

add listener

getRowCount()

getColumnCount()

getValueAt

```
57      catch (SQLException sqlex) {
58        sqlex.printStackTrace();
59      }
60
61      return null;
62    }
63
64    /** Return the column name at a specified column */
65    public String getColumnName(int column) {                    getColumnName()
66      try {
67        return rowSet.getMetaData().getColumnLabel(column + 1);
68      }
69      catch (SQLException ex) {
70        ex.printStackTrace();
71      }
72
73      return " ";
74    }
75
76    /** Implement rowSetChanged */
77    public void rowSetChanged(RowSetEvent e) {                   rowSetChanged
78      System.out.println("RowSet changed");
79      fireTableStructureChanged();
80    }
81
82    /** Implement rowChanged */
83    public void rowChanged(RowSetEvent e) {                      rowChanged
84      System.out.println("Row changed");
85      fireTableDataChanged();
86    }
87
88    /** Implement cursorMoved */
89    public void cursorMoved(RowSetEvent e) {                     cursorMoved
90      System.out.println("Cursor moved");
91    }
92  }
```

The `RowSetTableModel` class defines the `rowSet` property with get and set methods (lines 11–22). The `setRowSet` method sets a new `rowSet` in the model. The model becomes a listener for the `rowSet` (line 19) in response to the changes in the `rowSet`. The `fireTableStructureChanged()` method defined in `AbstractTableModel` is invoked to refill the model with the data in `rowSet` (line 20).

The `getRowCount()` method returns the number of rows in the `rowSet`. Invoking `rowSet.last()` moves the cursor to the last row (line 27), and `rowSet.getRow()` returns the row number (line 28).

The `getColumnCount()` method returns the number of columns in the `rowSet`. The number of the columns in the `rowSet` can be obtained from the meta data (line 41).

The `getValueAt(row, column)` method returns the cell value at the specified `row` and `column` (lines 52–62). The `getColumnName(column)` method returns the column name for the specified column (lines 65–74).

Note

The index of row and column in `JTable` is `0`-based. However, the index of row and column in `RowSet` is `1`-based.

index inconsistency

The `RowSetTableModel` implements the `RowSetListener` (lines 79–93). So, a `RowSetTableModel` can be a listener for `RowSet` events.

Now let us turn our attention to developing a useful utility that displays a row set in a `JTable`. As shown in Figure 38.4, you enter or select a JDBC driver and database, enter a username and a password, and specify a table name to connect the database and display the table contents in the `JTable`. You can then use the buttons *First*, *Next*, *Prior*, and *Last* to move the cursor to the first row, next row, previous row, and last row in the table, use the *Delete* button to delete a selected row, and use the *Commit* button to save the change in the database.

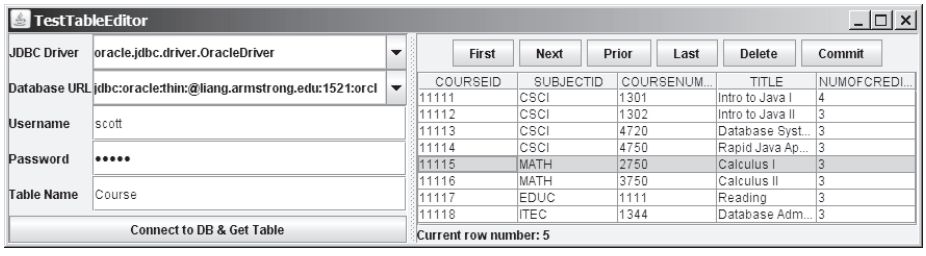


FIGURE 38.4 The program enables you to navigate the table and delete rows.

The status bar at the bottom of the window shows the current row in the row set. The cursor in the row set and the row in the `JTable` are synchronized. You can move the cursor by using the navigation buttons or by selecting a row in the `JTable`.

Define two classes: `TestTableEditor` (Listing 38.9) and `TableEditor` (Listing 38.10). `TestTableEditor` is the main class that enables the user to enter the database connection information and a table name. Once the database is connected, the table contents are displayed in an instance of `TableEditor`. The `TableEditor` class can be used to browse a table and modify a table.

LISTING 38.9 TestTableEditor.java

```
1  import javax.swing.*;
2  import java.awt.*;
3  import java.awt.event.*;
4  import javax.sql.RowSet;
5  import com.sun.rowset.CachedRowSetImpl;
6
7  public class  TestTableEditor extends JApplet {
                                                          drives
8    private JComboBox jcboDriver = new JComboBox(new String[] {
9      "sun.jdbc.odbc.JdbcOdbcDriver",
10     "com.mysql.jdbc.Driver",
11     "oracle.jdbc.driver.OracleDriver"
12   });
                                                          urls
13   private JComboBox jcboURL = new JComboBox(new String[] {
14     "jdbc:odbc:exampleMDBDataSource",
15     "jdbc:mysql://localhost/javabook",
16     "jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl"
17   });
18
                                                          UI components
19   private JButton jbtConnect =
20     new JButton("Connect to DB & Get Table");
21   private JTextField jtfUserName = new JTextField();
22   private JPasswordField jpfPassword = new JPasswordField();
23   private JTextField jtfTableName = new JTextField();
24   private TableEditor tableEditor1 = new TableEditor();
25   private JLabel jlblStatus = new JLabel();
26
```

```
27   /** Creates new form TestTableEditor */
28   public TestTableEditor() {
29     JPanel1 jPanel1 = new JPanel(new GridLayout(5, 0));                  create UI
30     jPanel1.add(jcboDriver);
31     jPanel1.add(jcboURL);
32     jPanel1.add(jtfUserName);
33     jPanel1.add(jpfPassword);
34     jPanel1.add(jtfTableName);
35
36     JPanel jPanel2 = new JPanel(new GridLayout(5, 0));
37     jPanel2.add(new JLabel("JDBC Driver"));
38     jPanel2.add(new JLabel("Database URL"));
39     jPanel2.add(new JLabel("Username"));
40     jPanel2.add(new JLabel("Password"));
41     jPanel2.add(new JLabel("Table Name"));
42
43     JPanel jPanel3 = new JPanel(new BorderLayout());
44     jPanel3.add(jbtConnect, BorderLayout.SOUTH);
45     jPanel3.add(jPanel2, BorderLayout.WEST);
46     jPanel3.add(jPanel1, BorderLayout.CENTER);
47     tableEditor1.setPreferredSize(new Dimension(400, 200));
48
49     jcboURL.setEditable(true);
50     jcboDriver.setEditable(true);
51
52     add(new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
53       jPanel3, tableEditor1), BorderLayout.CENTER);
54     add(jlblStatus, BorderLayout.SOUTH);
55
56     jbtConnect.addActionListener(new ActionListener() {
57       public void actionPerformed(ActionEvent evt) {
58         try {
59           // Connect to the database and create a rowset
60           Class.forName(((String)jcboDriver.getSelectedItem()).trim());  load driver
61           RowSet rowSet = new CachedRowSetImpl();                        create rowSet
62           rowSet.setUrl(((String)jcboURL.getSelectedItem()).trim());     set url
63           rowSet.setUsername(jtfUserName.getText().trim());              set username
64           rowSet.setPassword(new String(jpfPassword.getPassword()));     set password
65           rowSet.setCommand("select * from " +                           set command
66             jtfTableName.getText().trim());
67           rowSet.execute();                                              execute command
68           rowSet.beforeFirst();
69           tableEditor1.setRowSet(rowSet);                                rowSet to tableEditor1
70         }
71         catch (Exception ex) {
72           jlblStatus.setText(ex.toString());
73         }
74       }
75     });
76   }
77 }                                                                        main method omitted
```

When the user clicks the *Connect to DB & Get Table* button, a `CachedRowSet` is created (line 61). The url, username, password, and a command are set in the row set (lines 62–66). The row set is executed (line 67) and is plugged to the `TableEditor` (line 69).

LISTING 38.10 TableEditor.java

```
1 import javax.swing.*;
2 import javax.swing.table.*;
```

```
 3 import javax.swing.event.*;
 4 import java.awt.*;
 5 import java.awt.event.*;
 6 import javax.sql.*;
 7 import com.sun.rowset.CachedRowSetImpl;
 8
 9 public class TableEditor extends JPanel {
10   private JButton jbtFirst = new JButton("First");
11   private JButton jbtNext = new JButton("Next");
12   private JButton jbtPrior = new JButton("Prior");
13   private JButton jbtLast = new JButton("Last");
14   private JButton jbtDelete = new JButton("Delete");
15   private JButton jbtCommit = new JButton("Commit");
16   private JLabel jlblStatus = new JLabel();
17
18   // Table model, table selection model, table, rowset
19   private RowSetTableModel tableModel = new RowSetTableModel();
20   private DefaultListSelectionModel listSelectionModel =
21     new DefaultListSelectionModel();
22   private JTable jTable1 = new JTable();
23   private RowSet rowSet;
24
25   /** Set a new row set */
26   public void setRowSet(RowSet rowSet) {
27     this.rowSet = rowSet;
28     tableModel.setRowSet(rowSet);
29     jTable1.setModel(tableModel);
30
31     // Enable auto sort on columns
32     TableRowSorter<TableModel> sorter =
33       new TableRowSorter<TableModel>(tableModel);
34     jTable1.setRowSorter(sorter);
35   }
36
37   /** Create a TableEditor */
38   public TableEditor() {
39     JPanel jPanel1 = new JPanel();
40     jPanel1.add(jbtFirst);
41     jPanel1.add(jbtNext);
42     jPanel1.add(jbtPrior);
43     jPanel1.add(jbtLast);
44     jPanel1.add(jbtDelete);
45     jPanel1.add(jbtCommit);
46
47     setLayout(new BorderLayout());
48     add(jPanel1, BorderLayout.NORTH);
49     add(new JScrollPane(jTable1), BorderLayout.CENTER);
50     add(jlblStatus, BorderLayout.SOUTH);
51
52     // Set selection model for the table
53     jTable1.setSelectionModel(listSelectionModel);
54
55     // Register listeners
56     jbtFirst.addActionListener(new ActionListener() {
57       public void actionPerformed(ActionEvent evt) {
58         moveCursor("first");
59       }
60     });
61     jbtNext.addActionListener(new ActionListener() {
62       public void actionPerformed(ActionEvent evt) {
```

UI components

RowSetTableModel

selection model
JTable
rowSet

plug **rowSet**
plug **tableModel**

auto sort

create UI

plug selection model

move cursor

```
63          moveCursor("next");                                          move cursor
64        }
65      });
66      jbtPrior.addActionListener(new ActionListener() {
67        public void actionPerformed(ActionEvent evt) {
68          moveCursor("previous");                                      move cursor
69        }
70      });
71      jbtLast.addActionListener(new ActionListener() {
72        public void actionPerformed(ActionEvent evt) {
73          moveCursor("last");                                          move cursor
74        }
75      });
76      jbtDelete.addActionListener(new ActionListener() {
77        public void actionPerformed(ActionEvent evt) {
78          delete();                                                    delete row
79        }
80      });
81      jbtCommit.addActionListener(new ActionListener() {
82        public void actionPerformed(ActionEvent evt) {
83          try {
84            ((CachedRowSetImpl)rowSet).acceptChanges();                save changes
85          }
86          catch (java.sql.SQLException ex) {
87            ex.printStackTrace();
88          }
89        }
90      });
91      listSelectionModel.addListSelectionListener(
92        new ListSelectionListener() {
93        public void valueChanged(ListSelectionEvent e) {
94          handleSelectionValueChanged(e);
95        }
96      });
97    }
98
99    /* Delete a row */
100   private void delete() {
101     try {
102       // Delete the record from the database
103       int currentRow = rowSet.getRow();
104       rowSet.deleteRow();                                            delete row
105       if (rowSet.isAfterLast())
106         rowSet.last();
107       else if (rowSet.getRow() >= currentRow)
108         rowSet.absolute(currentRow);
109       setTableCursor();
110     }
111     catch (java.sql.SQLException ex) {
112       jlblStatus.setText(ex.toString());
113     }
114   }
115
116   /** Set cursor in the table and set the row number in the status */
117   private void setTableCursor() throws java.sql.SQLException {       synchronize table cursor
118     int row = rowSet.getRow();
119     listSelectionModel.setSelectionInterval(row - 1, row - 1);
120     jlblStatus.setText("Current row number: " + row);
121   }
122
```

move cursor

```
123   /** Move cursor to the specified location */
124   private void moveCursor(String whereToMove) {
125     try {
126       if (whereToMove.equals("first"))
127         rowSet.first();
128       else if (whereToMove.equals("next") && !rowSet.isLast())
129         rowSet.next();
130       else if (whereToMove.equals("previous") && !rowSet.isFirst())
131         rowSet.previous();
132       else if (whereToMove.equals("last"))
133         rowSet.last();
134       setTableCursor();
135     }
136     catch (java.sql.SQLException ex) {
137       jlblStatus.setText(ex.toString());
138     }
139   }
140
141   /** Handle the selection in the table */
142   private void handleSelectionValueChanged(ListSelectionEvent e) {
143     int selectedRow = jTable1.getSelectedRow();
144
145     try {
146       if (selectedRow != -1) {
147         rowSet.absolute(selectedRow + 1);
148         setTableCursor();
149       }
150     }
151     catch (java.sql.SQLException ex) {
152       jlblStatus.setText(ex.toString());
153     }
154   }
155 }
```

table cursor selection

The `setRowSet` method (lines 26–35) sets a new row set in `TableEditor`. The `rowSet` is plugged into the table model (line 29) and the table model is attached to the table (line 29). The code in lines 32–34 enables the column names to be sorted.

The handling of the navigation buttons *First*, *Next*, *Prior*, and *Last* is simply to invoke the methods `first()`, `next()`, `previous()`, and `last()` to move the cursor in the `rowSet` and (lines 126–133), at the same time, set the selected row in `JTable` by invoking `setTableCursor()` (line 134).

To implement the *Delete* button, invoke the `deleteRow()` method (line 104) to remove the row from the `rowSet`. After the row is removed, set the cursor to the next row in the `rowSet` (lines 105–108) and synchronize the cursor in the table (line 109).

Note that the `deleteRow()` method removes the row from the `CachedRowSet`. The *Commit* button actually saves the changes into the database (line 84).

To implement the handler for list-selection events on `jTable1`, set the cursor in the row set to match the row selected in `jTable1` (lines 142–154).

38.7 Storing and Retrieving Images in JDBC

A database can store not only numbers and strings, but also images. SQL3 introduced a new data type called BLOB (*B*inary *L*arge *OB*ject) for storing binary data, which can be used to store images. Another new SQL3 type is CLOB (*C*haracter *L*arge *OB*ject) for storing a large text in the character format. JDBC introduced the interfaces `java.sql.Blob` and `java.sql.Clob` to

support mapping for these new SQL types. You can use `getBlob`, `setBinaryStream`, `getClob`, `setBlob`, and `setClob`, to access SQL BLOB and CLOB values in the interfaces `ResultSet` and `PreparedStatement`.

To store an image into a cell in a table, the corresponding column for the cell must be of the BLOB type. For example, the following SQL statement creates a table whose type for the flag column is BLOB.

```
create table Country(name varchar(30), flag blob,
  description varchar(255));
```

In the preceding statement, the `description` column is limited to 255 characters, which is the upper limit for MySQL. For Oracle, the upper limit is 32,672 bytes. For a large character field, you can use the CLOB type for Oracle, which can store up to two GB characters. MySQL does not support CLOB. However, you can use BLOB to store a long string and convert binary data into characters.

Note

Access does not support the BLOB and CLOB types. supported?

To insert a record with images to a table, define a prepared statement like this one:

```
PreparedStatement pstmt = connection.prepareStatement(
  "insert into Country values(?, ?, ?)");
```

Images are usually stored in files. You may first get an instance of `InputStream` for an image file and then use the `setBinaryStream` method to associate the input stream with a cell in the table, as follows: store image

```
// Store image to the table cell
File file = new File(imageFilename);
InputStream inputImage = new FileInputStream(file);
pstmt.setBinaryStream(2, inputImage, (int)(file.length()));
```

To retrieve an image from a table, use the `getBlob` method, as shown below: retrieve image

```
// Store image to the table cell
Blob blob = rs.getBlob(1);
ImageIcon imageIcon = new ImageIcon(
  blob.getBytes(1, (int)blob.length()));
```

Listing 38.11 gives a program that demonstrates how to store and retrieve images in JDBC. The program first creates the table and stores data to it. Then the program retrieves the country names from the table and adds them to a combo box. When the user selects a name from the combo box, the country's flag and description are displayed, as shown in Figure 38.5.

FIGURE 38.5 The program enables you to retrieve data, including images, from a table and displays them.

LISTING 38.11 StoreAndRetrieveImage.java

```
1  import java.sql.*;
2  import java.io.*;
3  import javax.swing.*;
4  import java.awt.*;
5  import java.awt.event.*;
6
7  public class  StoreAndRetrieveImage extends JApplet {
8    // Connection to the database
9    private Connection connection;
10
11   // Statement for static SQL statements
12   private Statement stmt;
13
14   // Prepared statement
15   private PreparedStatement pstmt = null;
16   private DescriptionPanel descriptionPanel1
17     = new DescriptionPanel();
18
19   private JComboBox jcboCountry = new JComboBox();
20
21   /** Creates new form StoreAndRetrieveImage */
22   public StoreAndRetrieveImage() {
23     try {
24       connectDB(); // Connect to DB
25       storeDataToTable(); //Store data to the table (including image)
26       fillDataInComboBox(); // Fill in combo box
27       retrieveFlagInfo((String)(jcboCountry.getSelectedItem()));
28     }
29     catch (Exception ex) {
30       ex.printStackTrace();
31     }
32
33     jcboCountry.addItemListener(new ItemListener() {
34       public void itemStateChanged(ItemEvent evt) {
35         retrieveFlagInfo((String)(evt.getItem()));
36       }
37     });
38
39     add(jcboCountry, BorderLayout.NORTH);
40     add(descriptionPanel1, BorderLayout.CENTER);
41   }
42
43   private void connectDB() throws Exception {
44     // Load the driver
45     Class.forName("com.mysql.jdbc.Driver");
46     System.out.println("Driver loaded");
47
48     // Establish connection
49     connection = DriverManager.getConnection
50       ("jdbc:mysql://localhost/javabook");
51     System.out.println("Database connected");
52
53     // Create a statement for static SQL
```

load driver

connect database

```
54      stmt = connection.createStatement();                          create statement
55
56      // Create a prepared statement to retrieve flag and description
57      pstmt = connection.prepareStatement("select flag, description " +   prepare statement
58        "from Country where name = ?");
59    }
60
61    private void storeDataToTable() {
62      String[] countries = {"Canada", "UK", "USA", "Germany",             data to database
63        "India", "China"};
64
65      String[] imageFilenames = {"image/ca.gif", "image/uk.gif",
66        "image/us.gif", "image/germany.gif", "image/india.gif",
67        "image/china.gif"};
68
69      String[] descriptions = {"A text to describe Canadian " +
70        "flag is omitted", "British flag ...", "American flag ...",
71        "German flag ...", "Indian flag ...", "Chinese flag ..."};
72
73      try {
74        // Create a prepared statement to insert records
75        PreparedStatement pstmt = connection.prepareStatement(
76          "insert into Country values(?, ?, ?)");                     insert
77
78        // Store all predefined records
79        for (int i = 0; i < countries.length; i++) {
80          pstmt.setString(1, countries[i]);
81
82          // Store image to the table cell
83          java.net.URL url =                                          get image URL
84            this.getClass().getResource(imageFilenames[i]);
85          InputStream inputImage = url.openStream();
86          pstmt.setBinaryStream(2, inputImage,                        binary stream
87            (int)(inputImage.available()));
88
89          pstmt.setString(3, descriptions[i]);
90          pstmt.executeUpdate();
91        }
92
93        System.out.println("Table Country populated");
94      }
95      catch (Exception ex) {
96        ex.printStackTrace();
97      }
98    }
99
100   private void fillDataInComboBox() throws Exception {
101     ResultSet rs = stmt.executeQuery("select name from Country");
102     while (rs.next()) {
103       jcboCountry.addItem(rs.getString(1));                         fill combo box
104     }
105   }
106
107   private void retrieveFlagInfo(String name) {
```

```
108     try {
109       pstmt.setString(1, name);
110       ResultSet rs = pstmt.executeQuery();
111       if (rs.next()) {
112         Blob blob = rs.getBlob(1);
113         ImageIcon imageIcon = new ImageIcon(
114           blob.getBytes(1, (int)blob.length()));
115         descriptionPanel1.setImageIcon(imageIcon);
116         descriptionPanel1.setName(name);
117         String description = rs.getString(2);
118         descriptionPanel1.setDescription(description);
119       }
120     }
121     catch (Exception ex) {
122       System.err.println(ex);
123     }
124   }
125 }
```

set name

get image icon

set description

main method omitted

`DescriptionPanel` (line 16) is a component for displaying a country (name, flag, and description). This component was presented in Listing 17.6, DescriptionPanel.java.

The `storeDataToTable` method (lines 61–98) populates the table with data. The `fillDataInComboBox` method (lines 100–105) retrieves the country names and adds them to the combo box. The `retrieveFlagInfo(name)` method (lines 107–124) retrieves the flag and description for the specified country name.

Key Terms

BLOB type 38–27
CLOB type 38–27
batch mode 38–2
cached row set 38–15
row set 38–18
scrollable result set 38–2
updatable result set 38–11

Chapter Summary

1. This chapter developed a universal SQL client that can be used to access any local or remote relational database.

2. You can use the `addBatch(SQLString)` method to add SQL statements to a statement for batch processing.

3. You can create a statement to specify that the result set be scrollable and updatable. By default, the result set is neither of these.

4. The `RowSet` can be used to simplify Java database programming. A `RowSet` object is scrollable and updatable. A `RowSet` can fire a `RowSetEvent`.

5. You can store and retrieve image data in JDBC using the SQL BLOB type.

Review Questions

Section 38.3

38.1 What is batch processing in JDBC? What are the benefits of using batch processing?

38.2 How do you add an SQL statement to a batch? How do you execute a batch?

38.3 Can you execute a SELECT statement in a batch?

38.4 How do you know whether a JDBC driver supports batch updates?

Section 38.4

38.5 What is a scrollable result set? What is an updatable result set?

38.6 How do you create a scrollable and updatable `ResultSet`?

38.7 How do you know whether a JDBC driver supports a scrollable and updatable `ResultSet`?

Sections 38.5–38.6

38.8 What are the advantages of `RowSet`?

38.9 What are `JdbcRowSet` and `CachedRowSet`? What are the differences between them?

38.10 How do you create a `JdbcRowSet` and a `CachedRowSet`?

38.11 Can you scroll and update a `RowSet`? What method must be invoked to write the changes in a `CachedRowSet` to the database?

38.12 Describe the handlers in `RowSetListener`.

Section 38.7

38.13 How do you store images into a database?

38.14 How do you retrieve images from a database?

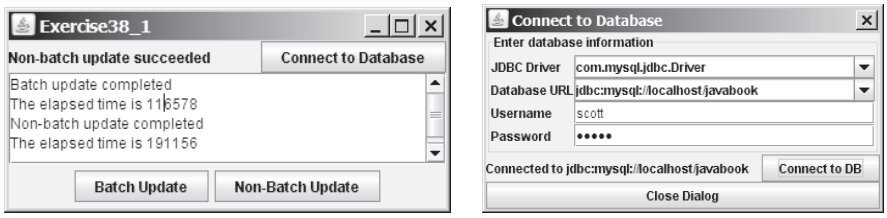
38.15 Does Oracle support the SQL3 BLOB type and CLOB type? What about MySQL and Access?

Programming Exercises

38.1* (*Batch update*) Write a program that inserts a thousand records to a database, and compare the performance with and without batch updates, as shown in Figure 38.6(a). Suppose the table is defined as follows:

```
create table Temp(num1 double, num2 double, num3 double)
```

Use the `Math.random()` method to generate random numbers for each record. Create a dialog box that contains `DBConnectionPanel`, discussed in Exercise 37.3. Use this dialog box to connect to the database. When you click the *Connect to Database* button in Figure 38.6(a), the dialog box in Figure 38.6(b) is displayed.



(a) (b)

FIGURE 38.6 The program demonstrates the performance improvements that result from using batch updates.

38.2** (*Scrollable result set*) Write a program that uses the buttons *First*, *Next*, *Prior*, *Last*, *Insert*, *Delete*, and *Update*, and modify a single record in the Address table, as shown in Figure 38.7.

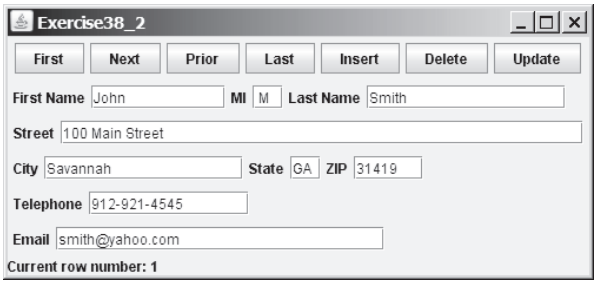


FIGURE 38.7 You can use the buttons to display and modify a single record in the Address table.

38.3** (*`ResultSetTableModel`*) Listing 38.8, RowSetTableModel.java, defines a table model for `RowSet`. Develop a new class named `ResultSetTableModel` for `ResultSet`. `ResultSetTableModel` extends `AbstractTableModel`. Write a test program that displays the `Course` table to a `JTable`, as shown in Figure 38.8. Enable autosort on columns.

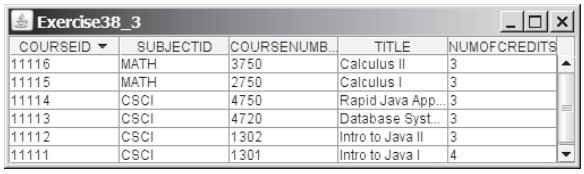


FIGURE 38.8 The Course table is displayed in a `JTable` using `ResultSetTableModel`.

38.4** (*Revising SQLClient.java*) Rewrite Listing 38.1, SQLClient.java, to display the query result in a `JTable`, as shown in Figure 38.9.

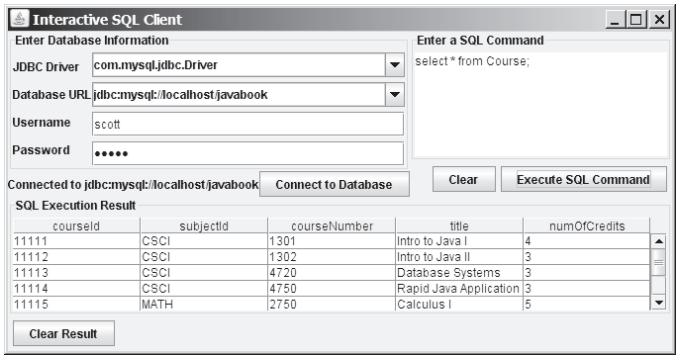


FIGURE 38.9 The query result is displayed in a `JTable`.

38.5* (*Editing table using `RowSet`*) Rewrite Listing 38.10 to add an *Insert* button to insert a new row and an *Update* button to update the row.

38.6* (*Displaying images from database*) Write a program that uses `JTable` to display the `Country` table created in Listing 38.11, StoreAndRetrieveImage.java, as shown in Figure 38.10.

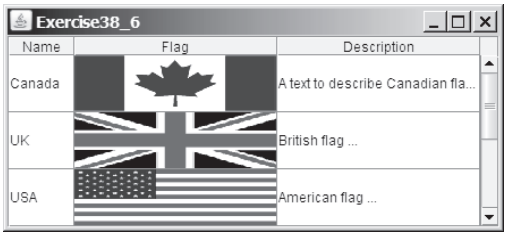


FIGURE 38.10 The Country table is displayed in a `JTable` instance.

38.7** (*Storing and retrieving images using `RowSet`*) Rewrite the example in Listing 38.11, StoreAndRetrieveImage.java, using `RowSet`.