CHAPTER 38

ADVANCED JAVA DATABASE PROGRAMMING

Objectives

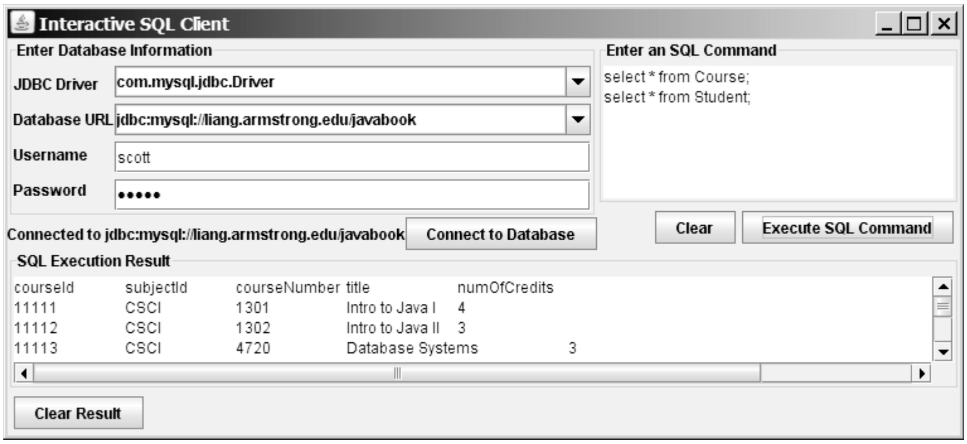
- To create a universal SQL client for accessing local or remote database (§38.2).
- To execute SQL statements in a batch mode (§38.3).
- To process updatable and scrollable result sets (§38.4).
- To simplify Java database programming using `RowSet` (§38.5).
- To create a custom table model for `RowSet`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



JDBC Driver: com.mysql.jdbc.Driver
Database URL: jdbc:mysql://liang.armstrong.edu/javabook
Username: scott
Password: •••••

Connected to jdbc:mysql://liang.armstrong.edu/javabook

```
select * from Course;
select * from Student;
```

```
courseId     subjectId     courseNumber title          numOfCredits
11111        CSCI          1301         Intro to Java I    4
11112        CSCI          1302         Intro to Java II   3
11113        CSCI          4720         Database Systems            3
```

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

LISTING 38.1 SQLClient.java

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

public class  SQLClient extends JApplet {
  // Connection to the database
  private Connection connection;

  // Statement to execute SQL commands
  private Statement statement;

  // Text area to enter SQL commands
  private JTextArea jtasqlCommand = new JTextArea();

```

connection

statement

```
  // Text area to display results from SQL commands
  private JTextArea jtaSQLResult = new JTextArea();

  // JDBC info for a database connection
  JTextField jtfUsername = new JTextField();
  JPasswordField jpfPassword = new JPasswordField();
  JComboBox jcboURL = new JComboBox(new String[] {
    "jdbc:mysql://liang.armstrong.edu/javabook",
    "jdbc:odbc:exampleMDBDataSource",
    "jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl"});
  JComboBox jcboDriver = new JComboBox(new String[] {
    "com.mysql.jdbc.Driver", "sun.jdbc.odbc.JdbcOdbcDriver",
    "oracle.jdbc.driver.OracleDriver"});

  JButton jbtExecuteSQL = new JButton("Execute SQL Command");
  JButton jbtClearSQLCommand = new JButton("Clear");
  JButton jbtConnectDB1 = new JButton("Connect to Database");
  JButton jbtClearSQLResult = new JButton("Clear Result");

  // Create titled borders
  Border titledBorder1 = new TitledBorder("Enter an SQL Command");
  Border titledBorder2 = new TitledBorder("SQL Execution Result");
  Border titledBorder3 = new TitledBorder(
    "Enter Database Information");

  JLabel jlblConnectionStatus = new JLabel("No connection now");

  /** Initialize the applet */
  public void init() {
    JScrollPane jScrollPane1 = new JScrollPane(jtasqlCommand);
    jScrollPane1.setBorder(titledBorder1);
    JScrollPane jScrollPane2 = new JScrollPane(jtaSQLResult);
    jScrollPane2.setBorder(titledBorder2);

    JPanel jPanel1 = new JPanel(new FlowLayout(FlowLayout.RIGHT));
    jPanel1.add(jbtClearSQLCommand);
    jPanel1.add(jbtExecuteSQL);

    JPanel jPanel2 = new JPanel();
    jPanel2.setLayout(new BorderLayout());
    jPanel2.add(jScrollPane1, BorderLayout.CENTER);
    jPanel2.add(jPanel1, BorderLayout.SOUTH);
    jPanel2.setPreferredSize(new Dimension(100, 100));

    JPanel jPanel3 = new JPanel();
    jPanel3.setLayout(new BorderLayout());
    jPanel3.add(jlblConnectionStatus, BorderLayout.CENTER);
    jPanel3.add(jbtConnectDB1, BorderLayout.EAST);

    JPanel jPanel4 = new JPanel();
    jPanel4.setLayout(new GridLayout(4, 1, 10, 5));
    jPanel4.add(jcboDriver);
    jPanel4.add(jcboURL);
    jPanel4.add(jtfUsername);
    jPanel4.add(jpfPassword);

    JPanel jPanel5 = new JPanel();
    jPanel5.setLayout(new GridLayout(4, 1));
    jPanel5.add(new JLabel("JDBC Driver"));
    jPanel5.add(new JLabel("Database URL"));
```

URLs

drivers

create UI

```
    jPanel5.add(new JLabel("Username"));
    jPanel5.add(new JLabel("Password"));

    JPanel jPanel6 = new JPanel();
    jPanel6.setLayout(new BorderLayout());
    jPanel6.setBorder(titledBorder3);
    jPanel6.add(jPanel4, BorderLayout.CENTER);
    jPanel6.add(jPanel5, BorderLayout.WEST);

    JPanel jPanel7 = new JPanel();
    jPanel7.setLayout(new BorderLayout());
    jPanel7.add(jPanel3, BorderLayout.SOUTH);
    jPanel7.add(jPanel6, BorderLayout.CENTER);

    JPanel jPanel8 = new JPanel();
    jPanel8.setLayout(new BorderLayout());
    jPanel8.add(jPanel2, BorderLayout.CENTER);
    jPanel8.add(jPanel7, BorderLayout.WEST);

    JPanel jPanel9 = new JPanel(new FlowLayout(FlowLayout.LEFT));
    jPanel9.add(jbtClearSQLResult);

    jcboURL.setEditable(true);
    jcboDriver.setEditable(true);

    add(jPanel8, BorderLayout.NORTH);
    add(jScrollPane2, BorderLayout.CENTER);
    add(jPanel9, BorderLayout.SOUTH);

    jbtExecuteSQL.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          executeSQL();
       }
    });
    jbtConnectDB1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        connectToDB();
      }
    });
    jbtClearSQLCommand.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        jtasqlCommand.setText(null);
      }
    });
    jbtClearSQLResult.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        jtaSQLResult.setText(null);
      }
    });
  }

  /** Connect to DB */
  private void connectToDB() {
    // 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());

```

execute SQL

connect database

clear command

clear result

```
    // Connection to the database
    try {
      Class.forName(driver);
      connection = DriverManager.getConnection(
        url, username, password);
      jlblConnectionStatus.setText("Connected to " + url);
    }
    catch (java.lang.Exception ex) {
      ex.printStackTrace();
    }
  }

  /** Execute SQL commands */
  private void executeSQL() {
    if (connection == null) {
      jtaSQLResult.setText("Please connect to a database first");
      return;
    }
    else {
      String sqlCommands = jtasqlCommand.getText().trim();
      String[] commands = sqlCommands.replace('\n', ' ').split(";");

      for (String aCommand: commands) {
        if (aCommand.trim().toUpperCase().startsWith("SELECT")) {
          processSQLSelect(aCommand);
        }
        else {
          processSQLNonSelect(aCommand);
        }
      }
    }
  }

  /** Execute SQL SELECT commands */
  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 = "";

      // Display column names
      for (int i = 1; i <= columnCount; i++) {
        row += resultSet.getMetaData().getColumnName(i) + "\t";
      }

      jtaSQLResult.append(row + '\n');

      while (resultSet.next()) {
        // Reset row to empty
        row = "";

        for (int i = 1; i <= columnCount; i++) {
          // A non-String column is converted to a string
```

load driver

connect database

process SQL select

process non-select

```
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();
```

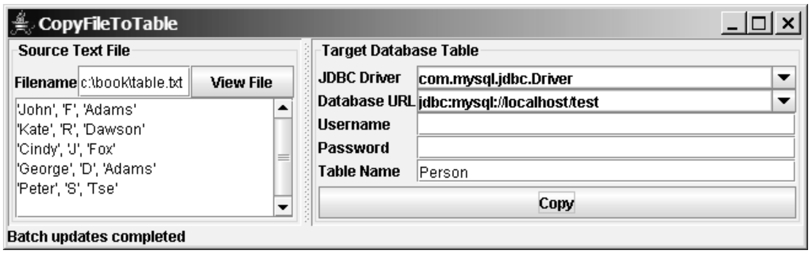
The executeBatch() method returns an array of counts, each of which counts the number of rows affected by the SQL command. The first count returns 0 because it is a DDL command. The other counts return 1 because only one row is affected.

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)
)
```



Filename c:\book\table.txt

```
'John', 'F', 'Adams'
'Kate', 'R', 'Dawson'
'Cindy', 'J', 'Fox'
'George', 'D', 'Adams'
'Peter', 'S', 'Tse'
```

JDBC Driver com.mysql.jdbc.Driver
Database URL jdbc:mysql://localhost/test
Username
Password
Table Name Person

Batch updates completed

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

```
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.sql.*;
import java.util.*;

public class  CopyFileToTable extends JFrame {
  // Text file info
  private JTextField jtfFilename = new JTextField();
  private JTextArea jtaFile = new JTextArea();

  // JDBC and table info
```

drivers

```
  private JComboBox jcboDriver = new JComboBox(new String[] {
    "com.mysql.jdbc.Driver", "sun.jdbc.odbc.JdbcOdbcDriver",
    "oracle.jdbc.driver.OracleDriver"});
```

URLs

```
  private JComboBox jcboURL = new JComboBox(new String[] {
    "jdbc:mysql://localhost/javabook",
    "jdbc:odbc:exampleMDBDataSource",
    "jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl"});
  private JTextField jtfUsername = new JTextField();
  private JPasswordField jtfPassword = new JPasswordField();
  private JTextField jtfTableName = new JTextField();

  private JButton jbtViewFile = new JButton("View File");
  private JButton jbtCopy = new JButton("Copy");
  private JLabel jlblStatus = new JLabel();

```

create UI

```
  public CopyFileToTable() {
    JPanel jPane1 = new JPanel();
    jPane1.setLayout(new BorderLayout());
    jPane1.add(new JLabel("Filename"), BorderLayout.WEST);
    jPane1.add(jbtViewFile, BorderLayout.EAST);
    jPane1.add(jtfFilename, BorderLayout.CENTER);

    JPanel jPane2 = new JPanel();
    jPane2.setLayout(new BorderLayout());
    jPane2.setBorder(new TitledBorder("Source Text File"));
    jPane2.add(jPane1, BorderLayout.NORTH);
    jPane2.add(new JScrollPane(jtaFile), BorderLayout.CENTER);

    JPanel jPane3 = new JPanel();
    jPane3.setLayout(new GridLayout(5, 0));
    jPane3.add(new JLabel("JDBC Driver"));
    jPane3.add(new JLabel("Database URL"));
    jPane3.add(new JLabel("Username"));
    jPane3.add(new JLabel("Password"));
    jPane3.add(new JLabel("Table Name"));

    JPanel jPane4 = new JPanel();
    jPane4.setLayout(new GridLayout(5, 0));
    jcboDriver.setEditable(true);
    jPane4.add(jcboDriver);
    jcboURL.setEditable(true);
    jPane4.add(jcboURL);
    jPane4.add(jtfUsername);
    jPane4.add(jtfPassword);
    jPane4.add(jtfTableName);

    JPanel jPane5 = new JPanel();
    jPane5.setLayout(new BorderLayout());
    jPane5.setBorder(new TitledBorder("Target Database Table"));
    jPane5.add(jbtCopy, BorderLayout.SOUTH);
    jPane5.add(jPane3, BorderLayout.WEST);
    jPane5.add(jPane4, BorderLayout.CENTER);

    add(jlblStatus, BorderLayout.SOUTH);
    add(new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
      jPane2, jPane5), BorderLayout.CENTER);

    jbtViewFile.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        showFile();                                              view file
      }
    });

    jbtCopy.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        try {
          copyFile();                                            to table
        }
        catch (Exception ex) {
          jlblStatus.setText(ex.toString());
        }
      }
    });
  }

  /** Display the file in the text area */
  private void showFile() {
    Scanner input = null;
    try {
      // Use a Scanner to read text from the file
      input = new Scanner(new File(jtfFilename.getText().trim()));

      // Read a line and append the line to the text area
      while (input.hasNext())
        jtaFile.append(input.nextLine() + '\n');
    }
    catch (FileNotFoundException ex) {
      System.out.println("File not found: " + jtfFilename.getText());
    }
    catch (IOException ex) {
      ex.printStackTrace();
    }
    finally {
      if (input != null) input.close();
    }
  }

  private void copyFile() throws Exception {
    // Load the JDBC driver
    Class.forName(((String)jcboDriver.getSelectedItem()).trim());   load driver
    System.out.println("Driver loaded");

    // Establish a connection
    Connection conn = DriverManager.getConnection                   connect database
      (((String)jcboURL.getSelectedItem()).trim(),
      jtfUsername.getText().trim(),
      String.valueOf(jtfPassword.getPassword()).trim());
    System.out.println("Database connected");

```

insert row

```
    // Read each line from the text file and insert it to the table
    insertRows(conn);
  }

  private void insertRows(Connection connection) {
    // Build the SQL INSERT statement
    String sqlInsert = "insert into " + jtfTableName.getText()
      + " values (";

    // Use a Scanner to read text from the file
    Scanner input = null;

    // Get file name from the text field
    String filename = jtfFilename.getText().trim();

    try {
      // Create a scanner
      input = new Scanner(new File(filename));

      // Create a statement
```

statement

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

      System.out.println("Driver major version? " +
        connection.getMetaData().getDriverMajorVersion());

      // Determine if batchUpdatesSupported is supported
      boolean batchUpdatesSupported = false;

      try {
        if (connection.getMetaData().supportsBatchUpdates()) {
```

batch

```
          batchUpdatesSupported = true;
          System.out.println("batch updates supported");
        }
        else {
          System.out.println("The driver " +
            "does not support batch updates");
        }
      }
      catch (UnsupportedOperationException ex) {
        System.out.println("The operation is not supported");
      }

      // Determine if the driver is capable of batch updates
      if (batchUpdatesSupported) {
        // Read a line and add the insert table command to the batch
        while (input.hasNext()) {
          statement.addBatch(sqlInsert + input.nextLine() + ")");
        }

        statement.executeBatch();

        jlblStatus.setText("Batch updates completed");
      }
      else {
        // Read a line and execute insert table command
        while (input.hasNext()) {
```

execute batch

```
          statement.executeUpdate(sqlInsert + input.nextLine() + ")");
        }

        jlblStatus.setText("Single row update completed");
```

```
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   }
199 }
```

main method omitted

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.

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 **get*Xxx*(int columnIndex)** or **get*Xxx*(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

```
import java.sql.*;

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

    // Establish a connection
    Connection connection = DriverManager.getConnection
      ("jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl",
       "scott", "tiger");
    connection.setAutoCommit(true);
    System.out.println("Database connected");

    // Get a new statement for the current connection
    Statement statement = connection.createStatement(
      ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);

    // Get ResultSet
    ResultSet resultSet = statement.executeQuery
      ("select state, capital from StateCapital");

    System.out.println("Before update ");
    displayResultSet(resultSet);

    // Update the second row
    resultSet.absolute(2); // Move cursor to the second row
    resultSet.updateString("state", "New S"); // Update the column
    resultSet.updateString("capital", "New C"); // Update the column
    resultSet.updateRow(); // Update the row in the data source

    // Insert after the second row
    resultSet.last();
    resultSet.moveToInsertRow(); // Move cursor to the insert row
    resultSet.updateString("state", "Florida");
    resultSet.updateString("capital", "Tallahassee");
    resultSet.insertRow(); // Insert the row
    resultSet.moveToCurrentRow(); // Move the cursor to the current row

    // Delete fourth row
    resultSet.absolute(4); // Move cursor to the 5th row
    resultSet.deleteRow(); // Delete the second row

    System.out.println("After update ");
    resultSet = statement.executeQuery
      ("select state, capital from StateCapital");
    displayResultSet(resultSet);

    // Close the connection
    resultSet.close();
  }

  private static void displayResultSet(ResultSet resultSet)
      throws SQLException {
    ResultSetMetaData rsMetaData = resultSet.getMetaData();
    resultSet.beforeFirst();
    while (resultSet.next()) {
      for (int i = 1; i <= rsMetaData.getColumnCount(); i++)
        System.out.printf("%-12s\t", resultSet.getObject(i));
      System.out.println();
    }
  }
}
```

get result set

move cursor

update row

move cursor

insert row

move cursor

delete row

close result set

display result set

```
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).

insert row

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

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

driver support

Note

Not all current drivers support scrollable and updatable result sets. The example is tested using Oracle ojdbc14 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.

driver support

Note

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

extends ResultSet

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. A concrete RowSet class can be used as a JavaBeans component in a visual GUI development environment such as NetBeans and JBuilder.

supported?

Note

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.

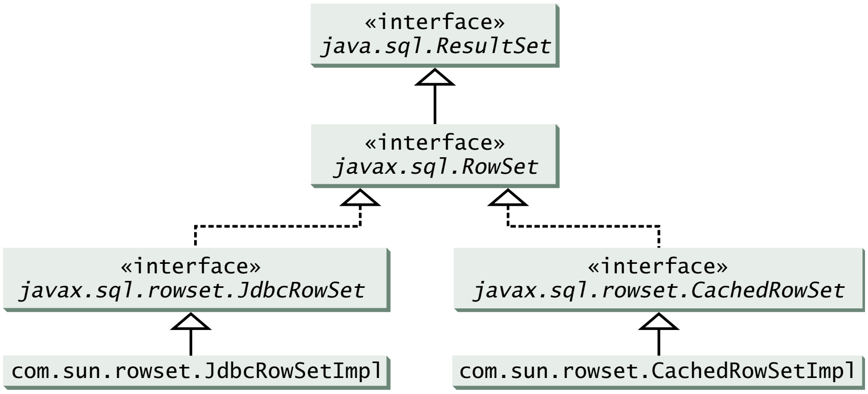

«interface»
java.sql.ResultSet
«interface»
javax.sql.RowSet
«interface»
javax.sql.rowset.JdbcRowSet
«interface»
javax.sql.rowset.CachedRowSet
com.sun.rowset.JdbcRowSetImpl
com.sun.rowset.CachedRowSetImpl

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

```
import java.sql.SQLException;
import javax.sql.RowSet;
import com.sun.rowset.*;

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

    // Create a row set
    RowSet rowSet = new JdbcRowSetImpl();

    // Set RowSet properties
    rowSet.setUrl("jdbc:mysql://localhost/javabook");
    rowSet.setUsername("scott");
```

load driver

create **RowSet**

set url

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() are 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.

obtain metadata

Tip

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

connected vs. disconnected

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");
```

```
    rowSet.setPassword("tiger");
    rowSet.setCommand("select * from Student where lastName = ? " +
     "and mi = ?");
    rowSet.setString(1, "Smith");
    rowSet.setString(2, "R");
    rowSet.execute();

    ResultSetMetaData rsMetaData = rowSet.getMetaData();
    for (int i = 1; i <= rsMetaData.getColumnCount(); i++)
      System.out.printf("%-12s\t", rsMetaData.getColumnName(i));
    System.out.println();

    // Iterate through the result and print the student names
    while (rowSet.next()) {
      for (int i = 1; i <= rsMetaData.getColumnCount(); i++)
        System.out.printf("%-12s\t", rowSet.getObject(i));
      System.out.println();
    }

    // Close the connection
    rowSet.close();
  }
}
```

SQL with parameters

set parameter

set parameter

execute

metadata

close connection

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

```
import java.sql.*;
import javax.sql.RowSet;
import com.sun.rowset.JdbcRowSetImpl;

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

    // Create a row set
    RowSet rowSet = new JdbcRowSetImpl();

    // Set RowSet properties
    rowSet.setUrl("jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl");
    rowSet.setUsername("scott");
    rowSet.setPassword("tiger");
    rowSet.setCommand("select state, capital from StateCapital");
    rowSet.execute();

    System.out.println("Before update ");
    displayRowSet(rowSet);

```

load driver

create a **RowSet**

set url

set username

set password

set SQL command

execute

display **rowSet**

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 **cursorSetChangeed(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

```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");
10     System.out.println("Driver loaded");
11
12     // Create a row set
13     RowSet rowSet = new CachedRowSetImpl();
14     rowSet.addRowSetListener(new RowSetListener() {
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();
34
35     rowSet.last(); // Cursor moved
36     rowSet.updateString("lastName", "Yao"); // Update column
37     rowSet.updateRow(); // Row updated
38
39     // Close the connection
40     rowSet.close();
41   }
42 }
```

load driver

create **RowSet**

register listener

cursor moved

row updated

38.6 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

```java
import java.sql.*;
import javax.sql.*;
import javax.swing.table.AbstractTableModel;

public class  RowSetTableModel extends AbstractTableModel
    implements RowSetListener {
  // RowSet for the result set
  private RowSet rowSet;

  /** Return the rowset */
  public RowSet getRowSet() {
    return rowSet;
  }

  /** Set a new rowset */
  public void setRowSet(RowSet rowSet) {
    if (rowSet != null) {
      rowSet.removeRowSetListener(this);
    }

    this.rowSet = rowSet;
    rowSet.addRowSetListener(this);
    fireTableStructureChanged();
  }

  /** Return the number of rows in the row set */
  public int getRowCount() {
    try {
      rowSet.last();
      return rowSet.getRow();
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }

    return 0;
  }

  /** Return the number of columns in the row set */
  public int getColumnCount() {
    try {
      if (rowSet != null) {
        return rowSet.getMetaData().getColumnCount();
      }
    }
    catch (SQLException ex) {
      ex.printStackTrace();
    }

    return 0;
  }

  /** Return value at the specified row and column */
  public Object getValueAt(int row, int column) {
    try {
      rowSet.absolute(row + 1);
      return rowSet.getObject(column + 1);
    }
```

rowSet

getRowSet

setRowSet

add listener

getRowCount()

getColumnCount()

getValueAt

```
59     catch (SQLException sqlex) {
60       sqlex.printStackTrace();
61     }
62 
63     return null;
64   }
65 
66   /** Return the column name at a specified column */
67   public String getColumnName(int column) {
68     try {
69       return rowSet.getMetaData().getColumnLabel(column + 1);
70     }
71     catch (SQLException ex) {
72       ex.printStackTrace();
73     }
74 
75     return " ";
76   }
77 
78   /** Implement rowSetChanged */
79   public void rowSetChanged(RowSetEvent e) {
80     System.out.println("RowSet changed");
81     fireTableStructureChanged();
82   }
83 
84   /** Implement rowChanged */
85   public void rowChanged(RowSetEvent e) {
86     System.out.println("Row changed");
87     fireTableDataChanged();
88   }
89 
90   /** Implement cursorMoved */
91   public void cursorMoved(RowSetEvent e) {
92     System.out.println("Cursor moved");
93   }
94 }
```

getColumnName()

rowSetChanged

rowChanged

cursorMoved

The RowSetTableModel class defines the rowSet property with get and set methods (lines 11–24). The setRowSet method sets a new rowSet in the model. The model becomes a listener for the rowSet (line 22) 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.

The getRowCount() method returns the number of rows in the rowSet. Invoking rowSet.last() moves the cursor to the last row (line 29), and rowSet.getRow() returns the row number (line 30).

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 43).

The getValueAt(row, column) method returns the cell value at the specified row and column (lines 54–64). The getColumnName(column) method returns the column name for the specified column (lines 67–76).

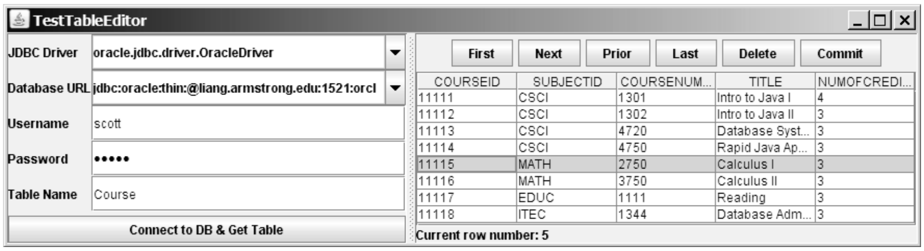
Note

The index of row and column in JTable is 0-based. 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.



JDBC Driver: oracle.jdbc.driver.OracleDriver

Database URL: jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl

Username: scott

Password: •••••

Table Name: Course

COURSEID	SUBJECTID	COURSENUM...	TITLE	NUMOFCREDI...
11111	CSCI	1301	Intro to Java I	4
11112	CSCI	1302	Intro to Java II	3
11113	CSCI	4720	Database Syst...	3
11114	CSCI	4750	Rapid Java Ap...	3
11115	MATH	2750	Calculus I	3
11116	MATH	3750	Calculus II	3
11117	EDUC	1111	Reading	3
11118	ITEC	1344	Database Adm...	3

Current row number: 5

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`.

Create 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

```java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.sql.RowSet;
import com.sun.rowset.CachedRowSetImpl;

public class  TestTableEditor extends JApplet {
```

drives

```java
  private JComboBox jcboDriver = new JComboBox(new String[] {
    "sun.jdbc.odbc.JdbcOdbcDriver",
    "com.mysql.jdbc.Driver",
    "oracle.jdbc.driver.OracleDriver"
  });
```

urls

```java
  private JComboBox jcboURL = new JComboBox(new String[] {
    "jdbc:odbc:exampleMDBDataSource",
    "jdbc:mysql://localhost/javabook",
    "jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl"
  });

```

UI components

```java
  private JButton jbtConnect =
    new JButton("Connect to DB & Get Table");
  private JTextField jtfUserName = new JTextField();
  private JPasswordField jpfPassword = new JPasswordField();
  private JTextField jtfTableName = new JTextField();
  private TableEditor tableEditor1 = new TableEditor();
  private JLabel jlblStatus = new JLabel();

```

```
27   /** Creates new form TestTableEditor */
28   public TestTableEditor() {
29     JPanel 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

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

UI components

RowSetTableModel

selection model

JTable

rowSet

plug **rowSet**

plug **tableModel**

auto sort

create UI

plug selection model

move cursor

```
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import javax.sql.*;
import com.sun.rowset.CachedRowSetImpl;

public class TableEditor extends JPanel {
  private JButton jbtFirst = new JButton("First");
  private JButton jbtNext = new JButton("Next");
  private JButton jbtPrior = new JButton("Prior");
  private JButton jbtLast = new JButton("Last");
  private JButton jbtDelete = new JButton("Delete");
  private JButton jbtCommit = new JButton("Commit");
  private JLabel jlblStatus = new JLabel();

  // Table model, table selection model, table, rowset
  private RowSetTableModel tableModel = new RowSetTableModel();
  private DefaultListSelectionModel listSelectionModel =
    new DefaultListSelectionModel();
  private JTable jTable1 = new JTable();
  private RowSet rowSet;

  /** Set a new row set */
  public void setRowSet(RowSet rowSet) {
    this.rowSet = rowSet;
    tableModel.setRowSet(rowSet);
    jTable1.setModel(tableModel);

    // Enable auto sort on columns
    TableRowSorter<TableModel> sorter =
      new TableRowSorter<TableModel>(tableModel);
    jTable1.setRowSorter(sorter);
  }

  /** Create a TableEditor */
  public TableEditor() {
    JPanel jPanel1 = new JPanel();
    jPanel1.add(jbtFirst);
    jPanel1.add(jbtNext);
    jPanel1.add(jbtPrior);
    jPanel1.add(jbtLast);
    jPanel1.add(jbtDelete);
    jPanel1.add(jbtCommit);

    setLayout(new BorderLayout());
    add(jPanel1, BorderLayout.NORTH);
    add(new JScrollPane(jTable1), BorderLayout.CENTER);
    add(jlblStatus, BorderLayout.SOUTH);

    // Set selection model for the table
    jTable1.setSelectionModel(listSelectionModel);

    // Register listeners
    jbtFirst.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        moveCursor("first");
      }
    });
    jbtNext.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        moveCursor("next");
      }
    });
    jbtPrior.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        moveCursor("previous");
      }
    });
    jbtLast.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        moveCursor("last");
      }
    });
    jbtDelete.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        delete();
      }
    });
    jbtCommit.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        try {
          ((CachedRowSetImpl)rowSet).acceptChanges();
        }
        catch (java.sql.SQLException ex) {
          ex.printStackTrace();
        }
      }
    });
    listSelectionModel.addListSelectionListener(
      new ListSelectionListener() {
      public void valueChanged(ListSelectionEvent e) {
        handleSelectionValueChanged(e);
      }
    });
  }

  /* Delete a row */
  private void delete() {
    try {
      // Delete the record from the database
      int currentRow = rowSet.getRow();
      rowSet.deleteRow();
      if (rowSet.isAfterLast())
        rowSet.last();
      else if (rowSet.getRow() >= currentRow)
        rowSet.absolute(currentRow);
      setTableCursor();
    }
    catch (java.sql.SQLException ex) {
      jlblStatus.setText(ex.toString());
    }
  }

  /** Set cursor in the table and set the row number in the status */
  private void setTableCursor() throws java.sql.SQLException {
    int row = rowSet.getRow();
    listSelectionModel.setSelectionInterval(row - 1, row - 1);
    jlblStatus.setText("Current row number: " + row);
  }

```

move cursor

move cursor

move cursor

delete row

save changes

delete row

synchronize table cursor

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 */
```

table cursor selection

```
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 }
```

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(?, ?, ?)");
```

store image

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:

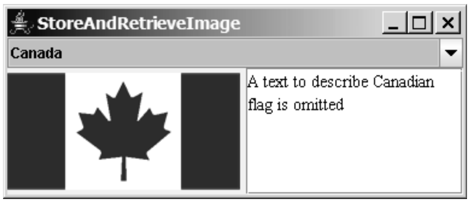
```
// Store image to the table cell
File file = new File(imageFilename);
InputStream inputImage = new FileInputStream(file);
pstmt.setBinaryStream(2, inputImage, (int)(file.length()));
```

retrieve image

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

```
// 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.



Canada

A text to describe Canadian flag is omitted

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

Listing 38.11 StoreAndRetrieveImage.java

```
import java.sql.*;
import java.io.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class  StoreAndRetrieveImage extends JApplet {
  // Connection to the database
  private Connection connection;

  // Statement for static SQL statements
  private Statement stmt;

  // Prepared statement
  private PreparedStatement pstmt = null;
  private DescriptionPanel descriptionPanel1
    = new DescriptionPanel();

  private JComboBox jcboCountry = new JComboBox();

  /** Creates new form StoreAndRetrieveImage */
  public StoreAndRetrieveImage() {
    try {
      connectDB(); // Connect to DB
      storeDataToTable(); //Store data to the table (including image)
      fillDataInComboBox(); // Fill in combo box
      retrieveFlagInfo((String)(jcboCountry.getSelectedItem()));
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }

    jcboCountry.addItemListener(new ItemListener() {
      public void itemStateChanged(ItemEvent evt) {
        retrieveFlagInfo((String)(evt.getItem()));
      }
    });

    add(jcboCountry, BorderLayout.NORTH);
    add(descriptionPanel1, BorderLayout.CENTER);
  }

  private void connectDB() throws Exception {
    // Load the driver
    Class.forName("com.mysql.jdbc.Driver");
    System.out.println("Driver loaded");

    // Establish connection
    connection = DriverManager.getConnection
      ("jdbc:mysql://localhost/javabook");
    System.out.println("Database connected");

    // Create a statement for static SQL
```

load driver

connect database

```
    stmt = connection.createStatement();

    // Create a prepared statement to retrieve flag and description
    pstmt = connection.prepareStatement("select flag, description " +
      "from Country where name = ?");
  }

  private void storeDataToTable() {
    String[] countries = {"Canada", "UK", "USA", "Germany",
      "India", "China"};

    String[] imageFilenames = {"image/ca.gif", "image/uk.gif",
      "image/us.gif", "image/germany.gif", "image/india.gif",
      "image/china.gif"};

    String[] descriptions = {"A text to describe Canadian " +
      "flag is omitted", "British flag ...", "American flag ...",
      "German flag ...", "Indian flag ...", "Chinese flag ..."};

    try {
      // Create a prepared statement to insert records
      PreparedStatement pstmt = connection.prepareStatement(
        "insert into Country values(?, ?, ?)");

      // Store all predefined records
      for (int i = 0; i < countries.length; i++) {
        pstmt.setString(1, countries[i]);

        // Store image to the table cell
        java.net.URL url =
          this.getClass().getResource(imageFilenames[i]);
        InputStream inputImage = url.openStream();
        pstmt.setBinaryStream(2, inputImage,
          (int)(inputImage.available()));

        pstmt.setString(3, descriptions[i]);
        pstmt.executeUpdate();
      }

      System.out.println("Table Country populated");
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  private void fillDataInComboBox() throws Exception {
    ResultSet rs = stmt.executeQuery("select name from Country");
    while (rs.next()) {
      jcboCountry.addItem(rs.getString(1));
    }
  }

  private void retrieveFlagInfo(String name) {
```

create statement

prepare statement

data to database

insert

get image URL

binary stream

fill combo box

set name

get image icon

set description

main method omitted

```
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 }
```

DescriptionPanel (line 16) is a component for displaying a country (name, flag, and description). This component was presented in Listing 16.6, 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	38–26	row set	38–15
CLOB type	38–26	scrollable result set	38–11
batch mode	38–7	updatable result set	38–11
cached row set	38–15		

Chapter Summary

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

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

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

- The RowSet can be used to simplify Java database programming. A RowSet object is scrollable and updatable. A RowSet can fire a RowSetEvent. A concrete RowSet class can be used as a JavaBeans component in a visual GUI development environment such as Eclipse, JBuilder, and NetBeans.

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

Review Questions

Section 38.3 Batch Processing

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 Scrollable and Updatable Result Set

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 Storing and Retrieving Images in JDBC

38.13 How do you store images into a database?

38.14 How do you retrieve images from a database?

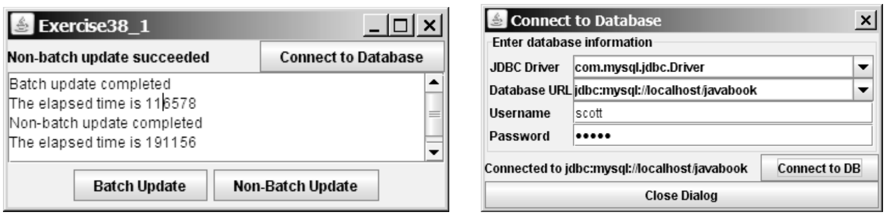
38.15 Does Oracle support the SQL3 BLOB type and CLOB type? What about MySQL and Access?

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.



Non-batch update succeeded

```
Batch update completed
The elapsed time is 116578
Non-batch update completed
The elapsed time is 191156
```

Enter database information

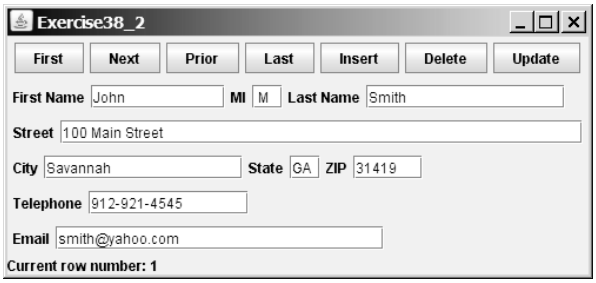
JDBC Driver: com.mysql.jdbc.Driver
Database URL: jdbc:mysql://localhost/javabook
Username: scott
Password: •••••

Connected to jdbc:mysql://localhost/javabook

(a) (b)

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

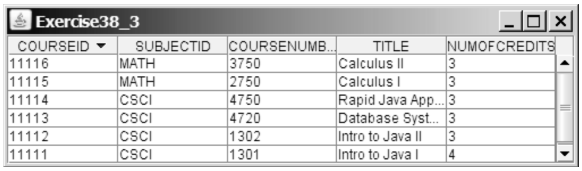
38.2** (*Scrollable result set*) Write a program that uses the buttons *First*, *Next*, *Prior*, *Last*, *Insert*, *Delete*, and *Update*, and modify a single record in the Address table, as shown in Figure 38.7.



First Name: John MI: M Last Name: Smith
Street: 100 Main Street
City: Savannah State: GA ZIP: 31419
Telephone: 912-921-4545
Email: smith@yahoo.com
Current row number: 1

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

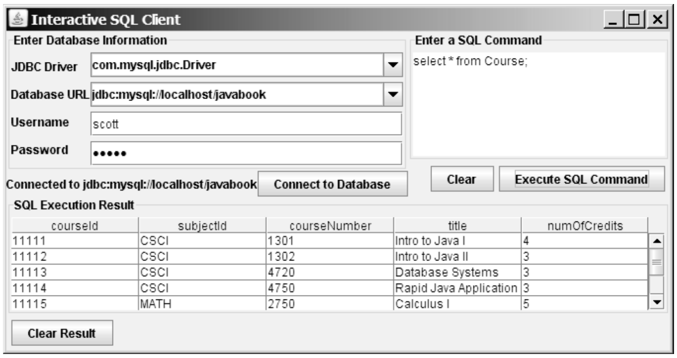
38.3** (*ResultSetTableModel*) Listing 38.8, RowSetTableModel.java, defines a table model for RowSet. Develop a new class named ResultSetTableModel for ResultSet. ResultSetTableModel extends AbstractTableModel. Write a test program that displays the Course table to a JTable, as shown in Figure 38.8.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



COURSEID ▾	SUBJECTID	COURSENUMB...	TITLE	NUMOFCREDITS
11116	MATH	3750	Calculus II	3
11115	MATH	2750	Calculus I	3
11114	CSCI	4750	Rapid Java App...	3
11113	CSCI	4720	Database Syst...	3
11112	CSCI	1302	Intro to Java II	3
11111	CSCI	1301	Intro to Java I	4

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.



JDBC Driver: com.mysql.jdbc.Driver

Database URL: jdbc:mysql://localhost/javabook

Username: scott

Password: •••••

select * from Course;

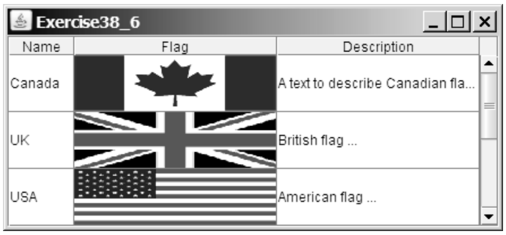
Connected to jdbc:mysql://localhost/javabook

courseId	subjectId	courseNumber	title	numOfCredits
11111	CSCI	1301	Intro to Java I	4
11112	CSCI	1302	Intro to Java II	3
11113	CSCI	4720	Database Systems	3
11114	CSCI	4750	Rapid Java Application	3
11115	MATH	2750	Calculus I	5

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.



Name	Flag	Description
Canada		A text to describe Canadian fla...
UK		British flag ...
USA		American flag ...

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`.