CHAPTER

35

Advanced Java Database Programming

Objectives

- To create a universal SQL client for accessing local or remote database (§35.2).
- To execute SQL statements in a batch mode (§35.3).
- To process updatable and scrollable result sets (§35.4).
- To simplify Java database programming using `RowSet` (§35.5).
- To store and retrieve images in JDBC (§35.6).

35.1 Introduction

Key Point

This chapter introduces advanced features for Java database programming.

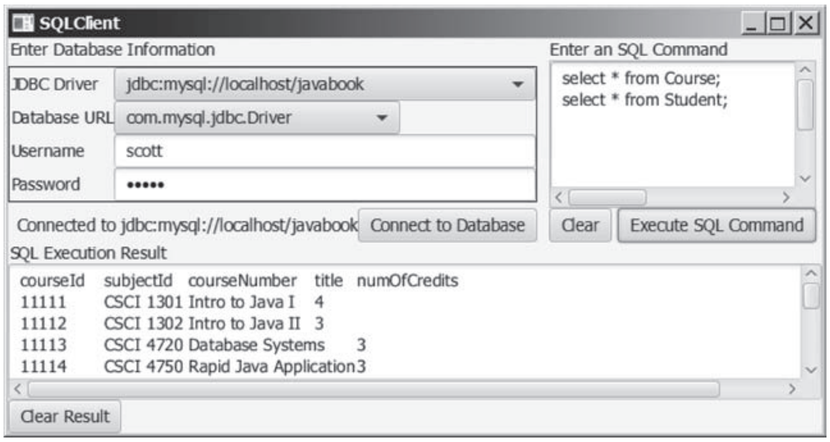
Chapter 34 introduced JDBC's basic features. This chapter covers its advanced features. You will learn how to develop a universal SQL client for accessing any local or remote relational database, learn how to execute statements in a batch mode to improve performance, learn scrollable result sets and 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.

35.2 A Universal SQL Client

Key Point

This section develops a universal SQL client for connecting and accessing any SQL database.

In Chapter 34, 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 35.1. 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 35.1 gives the program.



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

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

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 Application3
```

FIGURE 35.1 You can connect to any JDBC data source and execute SQL commands interactively.

LISTING 35.1 `SQLClient.java`

```java
import java.sql.*;
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.control.PasswordField;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

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

  // Statement to execute SQL commands
  private Statement statement;

  // Text area to enter SQL commands
  private TextArea tasqlCommand = new TextArea();

  // Text area to display results from SQL commands
  private TextArea taSQLResult = new TextArea();

  // DBC info for a database connection
  private TextField tfUsername = new TextField();
  private PasswordField pfPassword = new PasswordField();
  private ComboBox<String> cboURL = new ComboBox<>();
  private ComboBox<String> cboDriver = new ComboBox<>();

  private Button btExecuteSQL = new Button("Execute SQL Command");
  private Button btClearSQLCommand = new Button("Clear");
  private Button btConnectDB = new Button("Connect to Database");
  private Button btClearSQLResult = new Button("Clear Result");
  private Label lblConnectionStatus
    = new Label("No connection now");

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    cboURL.getItems().addAll(FXCollections.observableArrayList(
      "jdbc:mysql://localhost/javabook",
      "jdbc:mysql://liang.armstrong.edu/javabook",
      "jdbc:odbc:exampleMDBDataSource",
      "jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl"));
    cboURL.getSelectionModel().selectFirst();

    cboDriver.getItems().addAll(FXCollections.observableArrayList(
     "com.mysql.jdbc.Driver", "sun.jdbc.odbc.dbcOdbcDriver",
     "oracle.jdbc.driver.OracleDriver"));
    cboDriver.getSelectionModel().selectFirst();

    // Create UI for connecting to the database
    GridPane gridPane = new GridPane();
    gridPane.add(cboURL, 1, 0);
    gridPane.add(cboDriver, 1, 1);
    gridPane.add(tfUsername, 1, 2);
    gridPane.add(pfPassword, 1, 3);
    gridPane.add(new Label("JDBC Driver"), 0, 0);
    gridPane.add(new Label("Database URL"), 0, 1);
    gridPane.add(new Label("Username"), 0, 2);
    gridPane.add(new Label("Password"), 0, 3);

    HBox hBoxConnection = new HBox();
    hBoxConnection.getChildren().addAll(
      lblConnectionStatus, btConnectDB);
    hBoxConnection.setAlignment(Pos.CENTER_RIGHT);

    VBox vBoxConnection = new VBox(5);
    vBoxConnection.getChildren().addAll(
      new Label("Enter Database Information"),
      gridPane, hBoxConnection);

    gridPane.setStyle("-fx-border-color: black;");

    HBox hBoxSQLCommand = new HBox(5);
    hBoxSQLCommand.getChildren().addAll(
      btClearSQLCommand, btExecuteSQL);
    hBoxSQLCommand.setAlignment(Pos.CENTER_RIGHT);

    BorderPane borderPaneSqlCommand = new BorderPane();
    borderPaneSqlCommand.setTop(
      new Label("Enter an SQL Command"));
    borderPaneSqlCommand.setCenter(
      new ScrollPane(tasqlCommand));
    borderPaneSqlCommand.setBottom(
      hBoxSQLCommand);

    HBox hBoxConnectionCommand = new HBox(10);
    hBoxConnectionCommand.getChildren().addAll(
      vBoxConnection, borderPaneSqlCommand);

    BorderPane borderPaneExecutionResult = new BorderPane();
    borderPaneExecutionResult.setTop(
      new Label("SQL Execution Result"));
    borderPaneExecutionResult.setCenter(taSQLResult);
    borderPaneExecutionResult.setBottom(btClearSQLResult);

    BorderPane borderPane = new BorderPane();
    borderPane.setTop(hBoxConnectionCommand);
    borderPane.setCenter(borderPaneExecutionResult);

    // Create a scene and place it in the stage
    Scene scene = new Scene(borderPane, 670, 400);
    primaryStage.setTitle("SQLClient"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    btConnectDB.setOnAction(e -> connectToDB());
    btExecuteSQL.setOnAction(e -> executeSQL());
    btClearSQLCommand.setOnAction(e -> tasqlCommand.setText(null));
    btClearSQLResult.setOnAction(e -> taSQLResult.setText(null));
  }

  /** Connect to DB */
  private void connectToDB() {
    // Get database information from the user input
    String driver = cboDriver
      .getSelectionModel().getSelectedItem();
    String url = cboURL.getSelectionModel().getSelectedItem();
    String username = tfUsername.getText().trim();
    String password = pfPassword.getText().trim();

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

    /** Execute SQL commands */
    private void executeSQL() {
      if (connection == null) {
        taSQLResult.setText("Please connect to a database first");
        return;
      }
      else {
        String sqlCommands = tasqlCommand.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";
        }

        taSQLResult.appendText(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
            row += resultSet.getString(i) + "\t";
          }

          taSQLResult.appendText(row + '\n');
        }
      }
      catch (SQLException ex) {
        taSQLResult.setText(ex.toString());
      }
    }

```

```
200   /** Execute SQL DDL, and modification commands */
201   private void processSQLNonSelect(String sqlCommand) {
202     try {
203       // Get a new statement for the current connection
204       statement = connection.createStatement();
205
206       // Execute a non-SELECT SQL command
207       statement.executeUpdate(sqlCommand);
208
209       taSQLResult.setText("SQL command executed");
210     }
211     catch (SQLException ex) {
212       taSQLResult.setText(ex.toString());
213     }
214   }
215 }
```

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 122–140).

When the user clicks the *Execute SQL Command* button, the `executeSQL()` method is invoked (lines 143–161) 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 153–158). If the command is a SELECT query, the `processSQLSelect` method is invoked (lines 164–198). This method uses the `executeQuery` method (line 170) to obtain the query result. The result is displayed in the text area `jtaSQLResult` (line 181). If the command is a non-SELECT query, the `processSQLNonSelect()` method is invoked (lines 201–214). This method uses the `executeUpdate` method (line 207) to execute the SQL command.

The `getMetaData` method (lines 173, 178) in the `ResultSet` interface is used to obtain an instance of `ResultSetMetaData`. The `getColumnCount` method (line 173) returns the number of columns in the result set, and the `getColumnName(i)` method (line 178) returns the column name for the *i*th column.

35.3 Batch Processing

Key Point

You can send a batch of SQL statements to the database for execution at once to improve efficiency.

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:

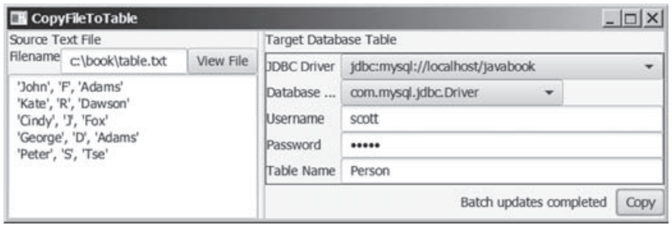
```
Statement statement = connection.createStatement();

// 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 35.2. 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 35.2 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: jdbc:mysql://localhost/javabook
Database ...: com.mysql.jdbc.Driver
Username: scott
Password: •••••
Table Name: Person

Batch updates completed

FIGURE 35.2 The `CopyFileToTable` utility copies text files to database tables.

Listing 35.2 gives the solution to the problem.

LISTING 35.2 CopyFileToTable.java

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.sql.*;
import java.util.Scanner;
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.control.PasswordField;
import javafx.scene.control.SplitPane;
```

```java
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class CopyFileToTable extends Application {
  // Text file info
  private TextField tfFilename = new TextField();
  private TextArea taFile = new TextArea();

  // JDBC and table info
  private ComboBox<String> cboURL = new ComboBox<>();
  private ComboBox<String> cboDriver = new ComboBox<>();
  private TextField tfUsername = new TextField();
  private PasswordField pfPassword = new PasswordField();
  private TextField tfTableName = new TextField();

  private Button btViewFile = new Button("View File");
  private Button btCopy = new Button("Copy");
  private Label lblStatus = new Label();

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    cboURL.getItems().addAll(FXCollections.observableArrayList(
      "jdbc:mysql://localhost/javabook",
      "jdbc:mysql://liang.armstrong.edu/javabook",
      "jdbc:odbc:exampleMDBDataSource",
      "jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl"));
    cboURL.getSelectionModel().selectFirst();

    cboDriver.getItems().addAll(FXCollections.observableArrayList(
      "com.mysql.jdbc.Driver", "sun.jdbc.odbc.dbcOdbcDriver",
      "oracle.jdbc.driver.OracleDriver"));
    cboDriver.getSelectionModel().selectFirst();

    // Create UI for connecting to the database
    GridPane gridPane = new GridPane();
    gridPane.add(new Label("JDBC Driver"), 0, 0);
    gridPane.add(new Label("Database URL"), 0, 1);
    gridPane.add(new Label("Username"), 0, 2);
    gridPane.add(new Label("Password"), 0, 3);
    gridPane.add(new Label("Table Name"), 0, 4);
    gridPane.add(cboURL, 1, 0);
    gridPane.add(cboDriver, 1, 1);
    gridPane.add(tfUsername, 1, 2);
    gridPane.add(pfPassword, 1, 3);
    gridPane.add(tfTableName, 1, 4);

    HBox hBoxConnection = new HBox(10);
    hBoxConnection.getChildren().addAll(lblStatus, btCopy);
    hBoxConnection.setAlignment(Pos.CENTER_RIGHT);

    VBox vBoxConnection = new VBox(5);
    vBoxConnection.getChildren().addAll(
      new Label("Target Database Table"),
      gridPane, hBoxConnection);

    gridPane.setStyle("-fx-border-color: black;");

    BorderPane borderPaneFileName = new BorderPane();
    borderPaneFileName.setLeft(new Label("Filename"));
    borderPaneFileName.setCenter(tfFilename);
    borderPaneFileName.setRight(btViewFile);

    BorderPane borderPaneFileContent = new BorderPane();
    borderPaneFileContent.setTop(borderPaneFileName);
    borderPaneFileContent.setCenter(taFile);

    BorderPane borderPaneFileSource = new BorderPane();
    borderPaneFileSource.setTop(new Label("Source Text File"));
    borderPaneFileSource.setCenter(borderPaneFileContent);

    SplitPane sp = new SplitPane();
    sp.getItems().addAll(borderPaneFileSource, vBoxConnection);

    // Create a scene and place it in the stage
    Scene scene = new Scene(sp, 680, 230);
    primaryStage.setTitle("CopyFileToTable"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    btViewFile.setOnAction(e -> showFile());
    btCopy.setOnAction(e -> {
      try {
        copyFile();
      }
      catch (Exception ex) {
        lblStatus.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(tfFilename.getText().trim()));

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

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

```java

    // Establish a connection
    Connection conn = DriverManager.getConnection(
      cboURL.getSelectionModel().getSelectedItem().trim(),
      tfUsername.getText().trim(),
      String.valueOf(pfPassword.getText()).trim());
    System.out.println("Database connected");

    // 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 " + tfTableName.getText()
      + " values (";

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

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

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

      // Create a 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()) {
          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();

        lblStatus.setText("Batch updates completed");
      }
```

```
198        else {
199          // Read a line and execute insert table command
200          while (input.hasNext()) {
201            statement.executeUpdate(sqlInsert + input.nextLine() + ")");
202          }
203
204          lblStatus.setText("Single row update completed");
205        }
206      }
207      catch (SQLException ex) {
208        System.out.println(ex);
209      }
210      catch (FileNotFoundException ex) {
211        System.out.println("File not found: " + filename);
212      }
213      finally {
214        if (input != null) input.close();
215      }
216    }
217  }
```

The `insertRows` method (lines 149–216) uses the batch updates to submit SQL INSERT commands to the database for execution, if the driver supports batch updates. Lines 174–181 check whether the driver supports batch updates. If the driver does not support the operation, an `UnsupportedOperationException` exception will be thrown (line 183) 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 35.1, you will write a program to insert a thousand records to a database and compare the performance with and without batch updates.

Check Point

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

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

35.3.3 Can you execute a SELECT statement in a batch?

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

35.4 Scrollable and Updatable Result Set

Key Point

You can use scrollable and updatable result set to move the cursor anywhere in the result set to perform insertion, deletion, and update.

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*.

A more powerful way of accessing database is to use a scrollable and updatable result, which enables you to 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:

- 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.

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);
```

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 35.3 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 35.3 ScrollUpdateResultSet.java

```
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");

    // Connect to a database
    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 last 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();
    }
  }
}
```

```
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
```

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

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).

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

Note

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

The program may not work due to an issue in the Oracle JDBC driver if lines 22–23 are replaced by

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

Check Point

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

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

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

35.5 RowSet, JdbcRowSet, and CachedRowSet

The RowSet interface can be used to simplify database programming.

Key Point

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.

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

35.5.1 RowSet Basics

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. Oracle has provided the reference implementation JdbcRowSetImpl for JdbcRowSet and CachedRowSetImpl for CachedRowSet. Figure 35.3 shows the relationship of these components.

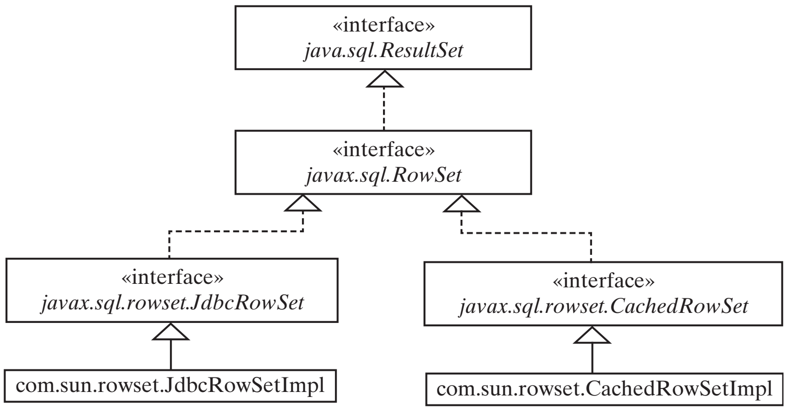

«interface»
java.sql.ResultSet
«interface»
javax.sql.RowSet
«interface»
javax.sql.rowset.JdbcRowSet
«interface»
javax.sql.rowset.CachedRowSet
com.sun.rowset.JdbcRowSetImpl
com.sun.rowset.CachedRowSetImpl

Figure 35.3 The JdbcRowSetImpl and CachedRowSetImpl are concrete implementations of RowSet.

The RowSet interface contains the JavaBeans properties with getter and setter methods. You can use the setter methods to set a new url, username, password, and command for an SQL statement. Using a RowSet, Listing 34.1 can be simplified, as shown in Listing 35.4.

Listing 35.4 SimpleRowSet.java

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

```

```
 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");
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 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`.

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

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.

35.5.2 RowSet for `PreparedStatement`

The discussion in §34.5, "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 35.5 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 35.5 `RowSetPreparedStatement.java`

```
1  import java.sql.*;
2  import javax.sql.RowSet;
3  import com.sun.rowset.*;
4
```

```
public class RowSetPreparedStatement {
  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");
    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();
  }
}
```

35.5.3 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 35.6 rewrites Listing 35.3 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 35.6 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("com.mysql.jdbc.Driver");
    System.out.println("Driver loaded");

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 state, capital from StateCapital");
20       rowSet.execute();
21
22       System.out.println("Before update ");
23       displayRowSet(rowSet);
24
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   }
```

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.

35.5.4 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 35.7 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 35.7 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");
10      System.out.println("Driver loaded");
11
12      // Create a row set
13      RowSet rowSet = new JdbcRowSetImpl();
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  }
```

Check Point

35.5.1 What are the advantages of `RowSet`?

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

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

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

35.5.5 Describe the handlers in `RowSetListener`.

35.6 Storing and Retrieving Images in JDBC

Key Point

You can store and retrieve images using 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 of characters. MySQL does not support CLOB. However, you can use BLOB to store a long string and convert binary data into characters.

Note

MS Access database does not support the BLOB and CLOB types.

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 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()));
```

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 35.8 gives a program that demonstrates how to store and retrieve images in JDBC. The program first creates the `Country` 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 35.4.

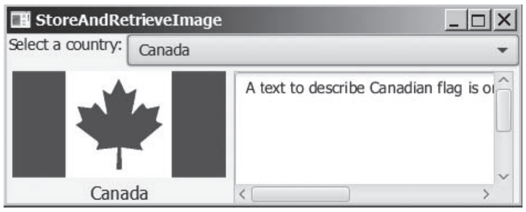


FIGURE 35.4 The program enables you to retrieve data, including images, from a table and displays them.

LISTING 35.8 StoreAndRetrieveImage.java

```java
import java.sql.*;
import java.io.*;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

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

  // Statement for static SQL statements
  private Statement stmt;

  // Prepared statement
  private PreparedStatement pstmt = null;
  private DescriptionPane descriptionPane
    = new DescriptionPane();

  private ComboBox<String> cboCountry = new ComboBox<>();

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    try {
      connectDB(); // Connect to DB
      storeDataToTable(); //Store data to the table (including image)
      fillDataInComboBox(); // Fill in combo box
      retrieveFlagInfo(cboCountry.getSelectionModel().getSelectedItem());
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }

    BorderPane paneForComboBox = new BorderPane();
    paneForComboBox.setLeft(new Label("Select a country: "));
    paneForComboBox.setCenter(cboCountry);
    cboCountry.setPrefWidth(400);
    BorderPane pane = new BorderPane();
    pane.setTop(paneForComboBox);
    pane.setCenter(descriptionPane);

    Scene scene = new Scene(pane, 350, 150);
    primaryStage.setTitle("StoreAndRetrieveImage");
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    cboCountry.setOnAction(e ->
      retrieveFlagInfo(cboCountry.getValue()));
  }

  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", "scott", "tiger");
    System.out.println("Database connected");

    // Create a statement for static SQL
    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",
      "Indian", "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();
```

```
103         }
104
105         System.out.println("Table Country populated");
106       }
107       catch (Exception ex) {
108         ex.printStackTrace();
109       }
110     }
111
112     private void fillDataInComboBox()  throws Exception {
113       ResultSet rs = stmt.executeQuery("select name from Country");
114       while (rs.next()) {
115         cboCountry.getItems().add(rs.getString(1));
116       }
117       cboCountry.getSelectionModel().selectFirst();
118     }
119
120     private void retrieveFlagInfo(String name) {
121       try {
122         pstmt.setString(1, name);
123         ResultSet rs = pstmt.executeQuery();
124         if (rs.next()) {
125           Blob blob = rs.getBlob(1);
126           ByteArrayInputStream in = new ByteArrayInputStream
127             (blob.getBytes(1, (int)blob.length()));
128           Image image = new Image(in);
129           ImageView imageView = new ImageView(image);
130           descriptionPane.setImageView(imageView);
131           descriptionPane.setTitle(name);
132           String description = rs.getString(2);
133           descriptionPane.setDescription(description);
134         }
135       }
136       catch (Exception ex) {
137         System.err.println(ex);
138       }
139     }
140   }
```

`DescriptionPane` (line 21) is a component for displaying a country (name, flag, and description). This component was presented in Listing 16.6, DescriptionPane.java.

The `storeDataToTable` method (lines 73–110) populates the table with data. The `fillDataInComboBox` method (lines 112–118) retrieves the country names and adds them to the combo box. The `retrieveFlagInfo(name)` method (lines 120–139) retrieves the flag and description for the specified country name.

Check Point

35.6.1 How do you store images into a database?

35.6.2 How do you retrieve images from a database?

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

Key Terms

BLOB type	35-20	row set	35-15
CLOB type	35-20	scrollable result set	35-2
batch mode	35-2	updatable result set	35-11
cached row set	35-15		

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.

Quiz

Answer the quiz for this chapter online at the book Companion Website.

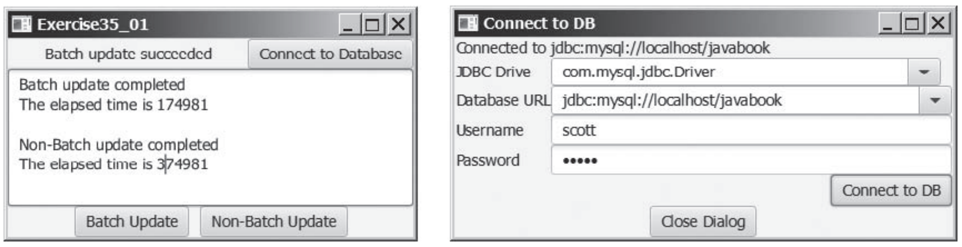
MyProgrammingLab™

Programming Exercises

***35.1** (*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 35.5a. 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 34.3. Use this dialog box to connect to the database. When you click the *Connect to Database* button in Figure 35.5a, the dialog box in Figure 35.5b is displayed.



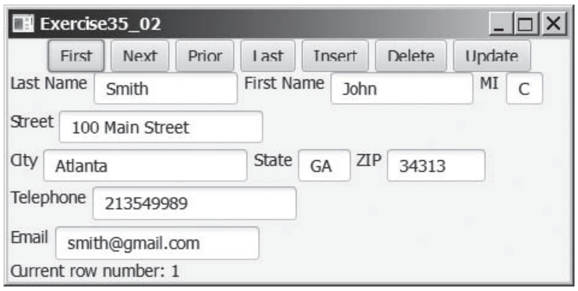
```
Batch update succeeded

Batch update completed
The elapsed time is 174981

Non-Batch update completed
The elapsed time is 374981
```

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

(a) (b)

Figure 35.5 The program demonstrates the performance improvements that result from using batch updates.

****35.2** (*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 35.6.



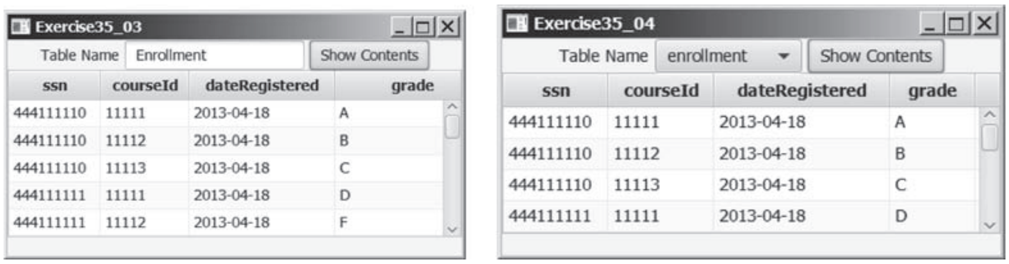
Last Name: Smith
First Name: John
MI: C
Street: 100 Main Street
City: Atlanta
State: GA
ZIP: 34313
Telephone: 213549989
Email: smith@gmail.com
Current row number: 1

FIGURE 35.6 You can use the buttons to display and modify a single record in the `Address` table.

The `Address` table is defined as follows:

```
create table Address (
  firstname varchar(25),
  mi char(1),
  lastname varchar(25),
  street varchar(40),
  city varchar(20),
  state varchar(2),
  zip varchar(5),
  telephone varchar(10),
  email varchar(30),
  primary key (firstname, mi, lastname)
);
```

***35.3** (*Display table contents*) Write a program that displays the content for a given table. As shown in Figure 35.7a, you enter a table and click the *Show Contents* button to display the table contents in a table view.



(a)

Table Name: Enrollment

ssn	courseId	dateRegistered	grade
444111110	11111	2013-04-18	A
444111110	11112	2013-04-18	B
444111110	11113	2013-04-18	C
444111111	11111	2013-04-18	D
444111111	11112	2013-04-18	F

(b)

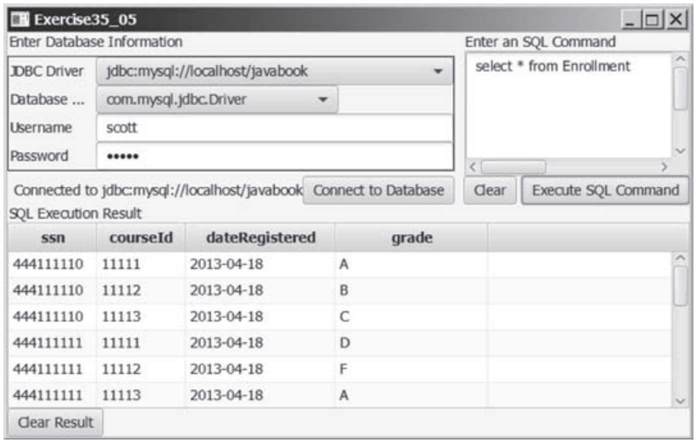
Table Name: enrollment

ssn	courseId	dateRegistered	grade
444111110	11111	2013-04-18	A
444111110	11112	2013-04-18	B
444111110	11113	2013-04-18	C
444111111	11111	2013-04-18	D

FIGURE 35.7 (a) Enter a table name to display the table contents. (b) Select a table name from the combo box to display its contents.

***35.4** (*Find tables and showing their contents*) Write a program that fills in table names in a combo box, as shown in Figure 35.7b. You can select a table from the combo box to display its contents in a table view.

****35.5** (*Revise SQLClient.java*) Rewrite Listing 35.1, SQLClient.java, to display the query result in a `TableView`, as shown in Figure 35.8.



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

select * from Enrollment

Connected to jdbc:mysql://localhost/javabook

ssn	courseId	dateRegistered	grade
444111110	11111	2013-04-18	A
444111110	11112	2013-04-18	B
444111110	11113	2013-04-18	C
444111111	11111	2013-04-18	D
444111111	11112	2013-04-18	F
444111111	11113	2013-04-18	A

FIGURE 35.8 The query result is displayed in a `TableView`.

***35.5** (*Populate Salary table)* Rewrite Programming Exercise 34.8 using a batch mode to improve performance.