Chapter 21
ADS and Java

Note: This chapter accompanies the book Advantage Database Server: A Developer's Guide, 2nd Edition, by Cary Jensen and Loy Anderson (2010, ISBN: 1453769978). For information on this book and on purchasing this book in various formats (print, e-book, etc), visit: http://www.JensenDataSystems.com/ADSBook10

This chapter provides you with examples of using the Advantage JDBC Driver with the JDBC classes to perform a wide range of common data-related tasks—using the Java language in general, and Borland's JBuilder in particular. As is the case with the other chapters in Part III, this discussion assumes that you are already familiar with Java programming.

ADS and Java

JDBC (Java database connectivity) is the core technology for accessing data from Java applications, applets, and servlets. Furthermore, using the JDBC Connector, available from Sun Microsystems, you can use this JDBC driver with any J2EE-compliant server. The Advantage JDBC Driver, named ADSDriver, is located in the com.extendedsystems.jdbc.advantage namespace. Once you have registered this driver and obtained a connection from the DriverManager, you access your Advantage data using the classes and interfaces of the java.sql namespace.

The Advantage JDBC Driver is a class 4 JDBC driver. Unlike class 1, class 2, and class 3 JDBC drivers, a class 4 driver requires no additional libraries, beyond the Java driver itself, to connect to the underlying data. With the Advantage JDBC Driver, the connection to ADS is accomplished using sockets. Unlike the other Advantage data access mechanisms, the Advantage JDBC Driver does not require the services of the Advantage Client Engine (ace32.dll and libace.so are the ACE libraries for Windows and Linux, respectively).

Note: Class 4 JDBC drivers connect directly to a server without requiring the installation of client drivers. Consequently, you can only use the Advantage JDBC Driver with ADS (since ALS is not a server).

The Advantage JDBC Driver communicates with ADS using TCP/IP port 6262 by default. If you need to communicate with ADS using a different port number on the server, you must change the server configuration. See the Advantage help for information

on how to configure your TCP/IP (transmission control protocol/Internet protocol) port number for the version of the ADS server that you are using.

Before you can use the Advantage JDBC Driver, you must install the adsjdbc.jar file and add it to your CLASSPATH environment variable. Java uses CLASSPATH to locate Java classes and other resources at runtime. The Advantage JDBC Driver installation will automatically install the JAR file. Depending on which environment you install the driver on, you may have to add the JAR file location to your CLASSPATH variable manually.

This chapter shows you how to access ADS using the Advantage JDBC Driver. This discussion is divided into three major sections. The first section describes common tasks, such as connecting to ADS, executing queries, and calling stored procedures. The second section shows you how to perform basic navigation with JDBC, and the third section demonstrates several basic administrative tasks, such as creating tables and granting rights to them.

The use of the Advantage JDBC Driver is demonstrated in this chapter using Borland's JBuilder, a popular Java IDE (integrated development environment). Figure 21-1 shows the AdsJava.jpx project opened in JBuilder 2006, with the public JFrame class (named MainFrame) displayed in the JBuilder designer.

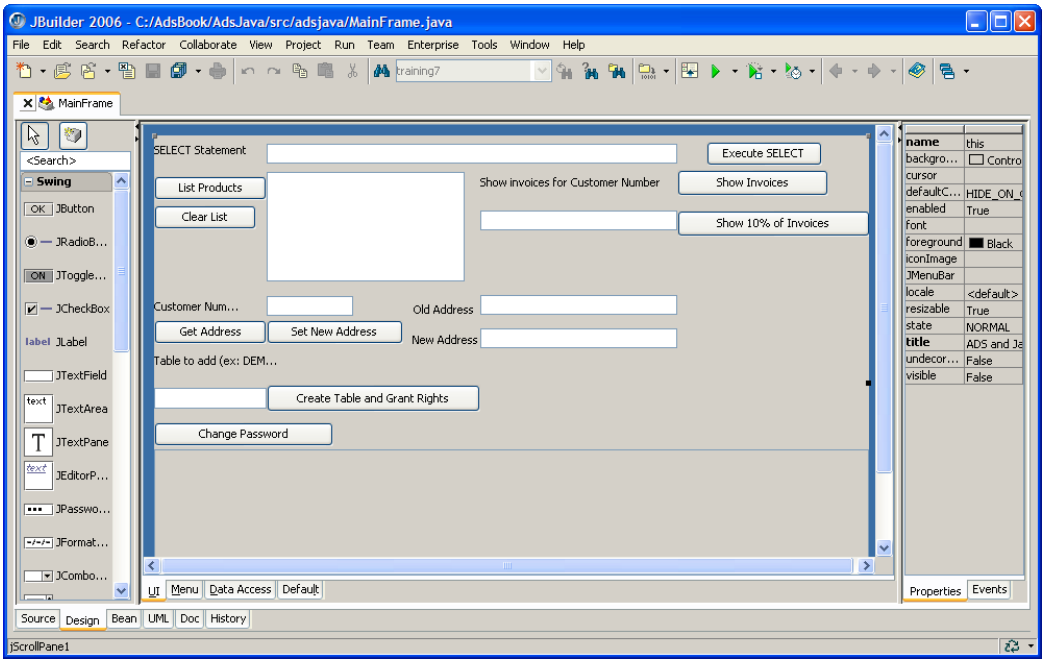

Figure 21-1: The MainFrame JFrame class in the JBuilder designer

Note: This same project can be created with almost any version of Java. Many earlier versions of JBuilder can also be used, as well as Java projects created with other development environments.

Code Download: The sample code in this chapter can be found in the JBuilder project named AdsJava.jpx, available with this book's code download (see Appendix A).

Even if you do not have a copy of JBuilder, you can still explore this project using the JDK (Java Development Kit) available from Sun Microsystems. Simply compile the two Java source files named Application1.java and MainFrame.java using javac.exe, the Java compiler. Once you have compiled these .java files into byte-code class files, launch the application by running the Application1.class file using java.exe, the Java runtime launcher. The Application1 class contains the public, static main method entry point.

Performing Basic Tasks with ADS and Java

This section describes some of the more common tasks that you can perform with Java and the Advantage JDBC Driver. These include connecting to a data dictionary, opening a table, executing a query, using a parameterized query, and executing a stored procedure.

Connecting to Data

You connect to a data dictionary or to a directory in which free tables are located by calling the getConnection method of the DriverManager. The getConnection method takes a connection string, which must be prefaced by the driver manager class that you want to get the connection for. For a connection to ADS, this prefix is jdbc:extendedsystems:advantage:.

Prior to calling getConnection, you must have instantiated the Advantage JDBC Driver. This is done by calling the forClass method of the Class class, passing the name of the Advantage JDBC Driver as an argument.

Because numerous event handlers associated with the MainFrame class use this connection, a variable of type Connection (a JDBC class) is declared in the MainFrame's class declaration, which places this variable in scope of all event handlers that need it. This variable declaration and several additional JDBC class variables that are used in two or more event handlers in this project are shown here:

```
public class MainFrame extends JFrame {
  Connection conn;
  Statement stmt;
  PreparedStatement prepStmt;
  //additional declarations
```

The Connection variable (conn) in the preceding segment is assigned a connection from a private method that is called from the MainFrame class constructor. This method, databaseInit, is shown in the following code segment:

```
private void databaseInit() throws Exception{
```

```
Class.forName
  ("com.extendedsystems.jdbc.advantage.ADSDriver");
  conn =
    DriverManager.getConnection("jdbc:extendedsystems:" +
    "advantage://server:6262/share/adsbook/"+
    "demodictionary.add;user=adsuser;password=password");
  stmt = conn.createStatement();
  prepStmt = conn.prepareStatement("SELECT * FROM INVOICE "+
    "WHERE [customer id] = ?" );
}
```

As you can see from the preceding method, forName is passed the name of the class of the Advantage JDBC Driver, which instantiates the driver. When the getConnection method of the DriverManager is called, it locates the instantiated driver by means of the prefix in the connection string.

In addition to containing the prefix for the Advantage JDBC Driver, this connection includes a URL (uniform resource locator) that points to the TCP/IP port on the machine named *server* where the data is located. This URL also includes an optional data location, identified by a share on that server (named *share* in this instance), and a qualified path to the data dictionary. Two additional parameters, the user name and password, are passed in this connection string as well.

Note: If an exception is raised when you attempt to connect, verify that your URL is correct and try again. You should also ensure that all clients on the same machine use a remote connection (since the Java driver only uses remote).

Because this connection string refers to the DemoDictionary data dictionary, and this dictionary requires logins, this particular connection string contains all of the essential parameters needed to connect to this database. Additional parameters could have been passed in this connection string as name/value pairs, where an equal sign separates the name and value. As you can see in the preceding connection string, when the connection string contains two or more name/value pairs, semicolons separate them. The full list of the optional connection string parameters is shown in Table 21-1.

Parameter	**Description**
Catalog	If the data directory or data dictionary path is not provided in the connection URL, set it to the qualified name of the data dictionary or the file location on the specified server where the free tables are located.
CharType	Identifies the character set used by the server. Can be set to ansi or oem. The default is ansi.
LockType	Identifies the type of locking to be used by ADS. Can be set to compatibility or proprietary. The default is proprietary.
Password	If the data dictionary requires login, use this parameter to submit the user's password.
QueryTimeout	The maximum number of second after which a SQL statement that has not completed will be aborted.

ShowDeleted	Set to true to include deleted records in DBF files. Set to false to suppress deleted records. The default is false.
TableType	Used to identify the type of table when connecting to free tables. This parameter can be set to adt, cdx, or ntx. The default is adt. This property is not used when you connect to a data dictionary.
User	If the data dictionary requires login, use this parameter to submit the user's user name.

Table 21-1: The Advantage Java JDBC Driver Connection String Parameters

Executing a Query

You can execute a query against ADS by calling any one of a number of methods of a java.sql.Statement instance, including execute, executeQuery, and executeUpdate. The execute method returns True if the statement returns a result, False if it does not, and throws an exception if the statement fails. The execute method is best when you do not know ahead of time if the statement returns a result set. Call executeQuery when you know that a result set will be returned, and executeUpdate when you know that one will not be returned.

The following event handler demonstrates the execution of a query that returns a result set. This event handler is associated with the Execute SELECT button (shown in Figure 21-1):

```
void executeSelect_actionPerformed(ActionEvent e) {
  try {
    ResultSet rs = stmt.executeQuery(selectText.getText());
    if (isRSEmpty(rs)) {
      JOptionPane.showMessageDialog(this,
       "No records in result set");
      return;
    }
    jTable1.setModel(new ResultTableModel(rs));
  }
  catch (Exception e1) {
    System.err.println( e1.getMessage());
  }
}
```

Since this is the first event handler from this project that we've inspected, there are two characteristics that need to be introduced—specifically, the isRSEmpty method and the use of the ResultTableModel class. Both of these are declared in the MainFrame.java file.

The isRSEmpty method is called by many of the event handlers in this application to determine whether or not there are records in the ResultSet returned by executeQuery. This method was added to the MainFrame class declaration as a public static method. The following is the implementation of this method:

```
public static boolean isRSEmpty(ResultSet rs) {
  try {
    return ! rs.first();
  }
```

```
  catch (Exception e1) {
    System.err.println( e1.getMessage());
    return false;
  }
}
```

The second item of interest is the class ResultTableModel. This class extends the abstract class AbstractTableModel, and it is used to create a model that can be used by the JTable class to display the contents of the result set. (Java swing classes employ a model-view architecture. The view is supplied by the visual component, and the model is responsible for handling the data.) At a minimum, ResultTableModel must override getColumnCount, getRowCount, and getValueAt. In this case, getColumnName is also overridden.

The following code implements the ResultTableModel class:

```
class ResultTableModel
  extends javax.swing.table.AbstractTableModel {
  Object obj [] [];
  int rows, columns;
  ResultSetMetaData rsMeta;

  public ResultTableModel (ResultSet rs) {
    try {
      if (rs == null) {
        rows = 0;
        columns = 0;
        obj = new Object[0][0];
        return;
      }
      rsMeta = rs.getMetaData();
      //get column count
      columns = rsMeta.getColumnCount();
      //calculate number of rows
      rows = 0;
      rs.first();
      do {
        rows++;
      } while (rs.next());
      //set array dimension
      obj = new Object [rows][columns];
      //load data
      rs.first();
      rows = 0;
      do {
        for (int j = 0; j <= (columns-1); j++) {
          obj[rows][j] = rs.getString(j+1);
        }
        rows++;
      } while (rs.next());
    } catch (Exception e1) {
    System.out.println(e1.getMessage());
    }
  }

    public int getColumnCount() {
```

```
            return columns;
        }

          public int getRowCount() {
              return rows;
          }

          public String getColumnName(int col)  {
            String res = "";
            if (rsMeta == null) {
              return res;
            }
            try {
              res = rsMeta.getColumnName(col+1);
            }
            catch (Exception e1) {
              System.out.println(e1.getMessage());
            }
            return res;
          }

          public Object getValueAt(int row, int col) {
              return obj[row][col];
          }
} //ResultTableModel class
```

As you can see in this code, the constructor of ResultTableModel is passed the ResultSet. This ResultSet is used to obtain a ResultSetMetaData object, which is then used to determine the number of columns in the ResultSet. This ResultSetMetaData object is also used to obtain the column names from within the getColumnName method.

Next, the ResultSet is navigated in order to count how many records the ResultSet contains. Finally, a two-dimensional array of Object is declared and populated with the rows and columns of the ResultSet.

Admittedly, this code is somewhat inefficient, in that it necessitates the retrieval of all of the records in the ResultSet, which is a time-consuming task when many records are involved. Consequently, this is not the type of TableModel that would be appropriate for every application. But for this sample Java project, it works just fine.

ResultTableModel is used to populate the JTable instance, a grid control that appears in the JFrame. Figure 21-2 shows this JTable populated with the results of a SQL SELECT statement.

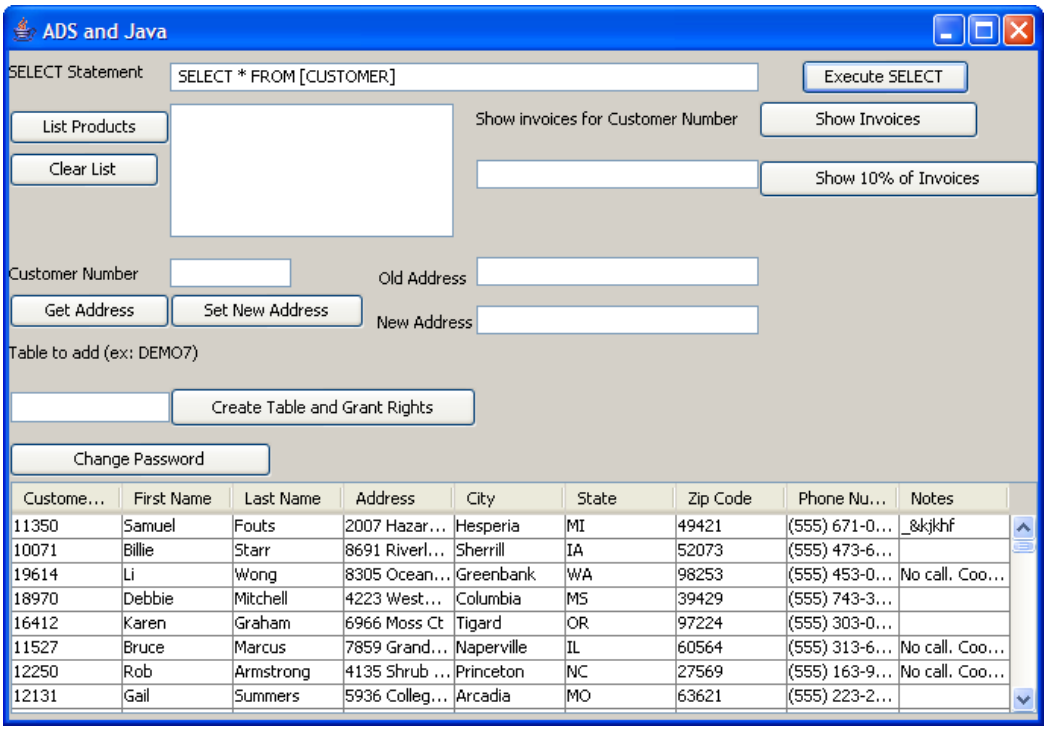

Figure 21-2: The JTable obtains its data from ResultTableModel

Using a Parameterized Query

Instead of using a Statement object, you use a PreparedStatement object when you need to execute a parameterized query. You can create a PreparedStatement object by calling the prepareStatement method of a Connection object, passing the parameterized query as an argument.

Before executing the PreparedStatement, you must call one of its setter methods for each parameter in the query. Which setter method you call depends on the data type of the parameter. If the parameter is a String, you call setString. On the other hand, if the parameter is an Integer, you call setInt.

The PreparedStatement was created in the databaseInit method shown earlier in this chapter. Data is bound to the single parameter and the query is executed from the following event handler, which is associated with the Show Invoices button (shown in Figure 21-1):

```
void showInvoiceBtn_actionPerformed(ActionEvent e) {
  try{
    prepStmt.setInt( 1,
      Integer.parseInt(paramText.getText()));
    ResultSet rs = prepStmt.executeQuery();
    if (isRSEmpty(rs)) {
      JOptionPane.showMessageDialog(this,
       "No records in result set");
```

```
      return;
    }
    jTable1.setModel(new ResultTableModel(rs));
  }
  catch (Exception e1) {
    System.err.println( e1.getMessage());
  }
}
```

Reading and Writing Data

You access individual columns in a ResultSet by calling one of its getter methods. All ResultSet getter methods are overloaded. You can identify a column either by ordinal position or by name.

Which getter method you call depends on the data type of the column you are reading. For example, you call getString in order to read a column containing text, and getBoolean to read a logical column.

If the result set is based on a live (dynamic) cursor, you can change its data and apply the change to the underlying Advantage table. You write to a column of a ResultSet by calling one of its setter methods. Like getter methods, ResultSet setter methods are overloaded, taking either the ordinal position of a field or the field name, in addition to the value you are writing to the field.

Once you have written to one or more fields of an updatable ResultSet record, you apply the changes to the underlying table by calling the ResultSet's updateRow method.

The following event handler, associated with the Get Address button (shown in Figure 21-1), demonstrates how to read a field from a ResultSet:

```
void getAddressBtn_actionPerformed(ActionEvent e) {
  PreparedStatement getCustStmt;
  if (custNoText.getText() == "") {
    System.out.println("Enter a customer ID");
    return;
  }
  try {
    getCustStmt = conn.prepareStatement(
      "SELECT * FROM CUSTOMER WHERE [customer id] = ?" );
    getCustStmt.setInt( 1,
      Integer.parseInt(custNoText.getText()));
    ResultSet rs = getCustStmt.executeQuery();
    if (isRSEmpty(rs)) {
      JOptionPane.showMessageDialog(this,
       "No records in result set");
      jTable1.setModel(new ResultTableModel(null));
      return;
    }
    oldAddressText.setText(rs.getString("Address"));
    jTable1.setModel(new ResultTableModel(rs));
  }
  catch (Exception e1) {
    System.err.println( e1.getMessage());
```

```
  }
}
```

The next event handler, associated with the Set New Address button (shown in Figure 21-1), demonstrates writing to a field and saving the change to ADS:

```
void setAddressBtn_actionPerformed(ActionEvent e) {
  PreparedStatement getCustStmt;
  if (custNoText.getText() == "") {
    System.out.println("Enter a customer ID");
    return;
  }
  try {
    getCustStmt = conn.prepareStatement(
      "SELECT * FROM CUSTOMER WHERE [customer id] = ?" );
    getCustStmt.setInt( 1,
      Integer.parseInt(custNoText.getText()));
    ResultSet rs = getCustStmt.executeQuery();
    if (isRSEmpty(rs)) {
      JOptionPane.showMessageDialog(this,
       "No records in result set");
      return;
    }
    rs.updateString("Address", newAddressText.getText());
    rs.updateRow();
  }
  catch (Exception e1) {
    System.err.println( e1.getMessage());
  }
}
```

Calling a Stored Procedure

Calling a stored procedure is no different than executing any other query. If your stored procedure does not require input parameters, you use a Statement instance. You use a PreparedStatement instance if there are one or more input parameters. If the stored procedure returns one or more records, you invoke the executeQuery method of the Statement or PreparedStatement object, and you invoke the execute or the executeUpdate methods when the stored procedure does not return records.

Invoking a stored procedure that takes one input parameter is demonstrated by the following code associated with the actionPerformed event handler for the Show 10% of Invoices button (shown in Figure 21-1). The stored procedure referenced in this code is the SQL stored procedure created in Chapter 7. If you did not create this stored procedure, but created one of the other stored procedures described in that chapter, substitute the name of the stored procedure object in your data dictionary in the EXECUTE PROCEDURE string, like this:

```
void callStoredProcBtn_actionPerformed(ActionEvent e) {
  PreparedStatement getCustStmt;
  if (custNoText.getText() == "") {
    System.out.println("Enter a customer ID");
    return;
  }
```

```
  try {
    getCustStmt = conn.prepareStatement(
      "EXECUTE PROCEDURE Get10PercentSQL( ? )" );
    getCustStmt.setInt( 1,
      Integer.parseInt(paramText.getText()));
    ResultSet rs = getCustStmt.executeQuery();
    if (isRSEmpty(rs)) {
      jTable1.setModel(new ResultTableModel(null));
      JOptionPane.showMessageDialog(this,
       "No records in result set");
      return;
    }
    jTable1.setModel(new ResultTableModel(rs));
  }
  catch (Exception e1) {
    JOptionPane.showMessageDialog(this,
      e1.getMessage());
  }
}
```

Navigational Actions with ADS and Java

Unlike Delphi and ADO-based Advantage applications, which support a wide range of navigational operations, JDBC supports only simple navigation. Specifically, the ResultSet class permits you to navigate forward through the records of the result set, and if the cursor is bidirectional, you can move forward and backward using methods with names such as first, next, last, and previous. The use of simple forward navigation is demonstrated in the following section.

Scanning a Result Set

Scanning is the process of sequentially reading every record in a result set. Although scanning is a common task, it is important to note that it necessarily requires the client application to retrieve all of the records in the result set. This is not a problem when few records are involved, but if a large number of records are being scanned, network resources may be taxed.

Tip: If you must scan a large number of records, implement the operation using a stored procedure. When ADS and the data are located on the same server, scanning from a stored procedure installed on ADS requires no network resources. Creating stored procedures is covered in Chapter 7, "Creating Stored Procedures."

The following code demonstrates scanning. It is associated with the actionPerformed event handler of the List Products button (shown in Figure 21-1), and it navigates the entire PRODUCTS table, assigning data from each record to the productList JListBox:

```
void listProductsBtn_actionPerformed(ActionEvent e) {
  DefaultListModel listModel =
    new javax.swing.DefaultListModel();
```

```
  try {
    ResultSet rs = stmt.executeQuery(
      "SELECT * FROM PRODUCTS");
    rs.first();
    do {
      listModel.addElement(rs.getString(1) + "     " +
        rs.getString(2));
    } while (rs.next());
    productList.setModel(listModel);
  }
  catch (Exception e1) {
    System.err.println( e1.getMessage());
  }
}
```

Note that the do-while loop in the preceding event handler could also have been written as follows using a while-do loop:

```
while (rs.next()) do {
  listModel.addElement(rs.getString(1) + "     " +
    rs.getString(2));
}
```

While the behavior of these two control structures is equivalent, there is a potential drawback to the second version, the while-do loop. Specifically, if the ResultSet has been navigated in any way prior to the while-do loop, the first record will be skipped. The do-while statement preceded by a call to the first method, by comparison, always processes every record in the ResultSet, whether or not the ResultSet has been navigated previously.

Administrative Operations with ADS and Java

While ADS requires little in the way of periodic maintenance to keep it running smoothly, many applications need to provide administrative functionality related to the management of users, groups, and other objects.

This section is designed to provide you with insight into exposing administrative functions in your client applications. Two related, yet different, operations are demonstrated here. In the first, a new table is added to the database and all groups are granted access rights to it. This operation requires that you establish an administrative connection, or a user connection with the appropriate GRANT rights. The second operation involves permitting individual users to modify their own passwords. Especially in the security-conscious world of modern database management, this feature is often considered an essential step to protecting data.

Creating a Table and Granting Rights to It

The AdsJava project permits a user to enter the name of a table that will be created in the data dictionary, after which all non-default groups will be granted rights to the table. This operation is demonstrated in the following event handler, which is associated with the actionPerformed event of the Create Table and Grant Rights button (shown in Figure 21-1):

```
void createTableBtn_actionPerformed(ActionEvent e) {
  boolean found = false;
  Connection adminconn;
  Statement adminstmt;
  Statement grantstmt;
  ResultSet rs;
  String tn = tableNameText.getText();
  //Check for semicolon hack
  if (! (tn.indexOf(";") == -1)) {
    JOptionPane.showMessageDialog(this,
      "Table name may not contain a semicolon");
    return;
  }
  if (tableNameText.getText().equals("")) {
     JOptionPane.showMessageDialog(this,
       "Please enter a table name");
    return;
  }
  try {
    adminconn = DriverManager.getConnection(
      "jdbc:extendedsystems:advantage://server:6262/share"+
      "/adsbook/"demodictionary.add;" +
      "user=adssys;password=password");
    adminstmt = adminconn.createStatement();
    rs =
      adminstmt.executeQuery(
        "SELECT NAME FROM system.tables");
    String tabName;
    if (! isRSEmpty(rs)) {
      rs.first();
      do {
        tabName = rs.getString("Name");
        if (tabName.equalsIgnoreCase(tn)) {
         found = true;
         break;
        }
      } while (rs.next());
    }
    if (found) {
      JOptionPane.showMessageDialog(this,
        "Table already exists. Cannot create");
      return;
    }
    adminstmt.executeUpdate("CREATE TABLE " + tn +
      "([Full Name] CHAR(30)," +
      "[Date of Birth] DATE," +
      "[Credit Limit] MONEY, " +
      "Active LOGICAL)");
    rs = adminstmt.executeQuery(
      "SELECT * FROM system.usergroups " +
         "WHERE NAME NOT LIKE 'DB:%'");
    if (isRSEmpty(rs)) {
      JOptionPane.showMessageDialog(this,
        "No groups to grant rights to");
      return;
    }
    grantstmt = adminconn.createStatement();
    rs.first();
```

```
    do {
      grantstmt.executeUpdate("GRANT ALL ON [" + tn + "]" +
        " TO [" +  rs.getString("Name") + "]");
    } while (rs.next());
    JOptionPane.showMessageDialog(this, "The " +
      tn + " table " +
      "has been created, with rights granted to all groups");
  } catch (Exception e1) {
    System.out.println(e1.getMessage());
  }
}
```

This event handler begins by verifying that the table name does not include a semicolon, which could be used to introduce a SQL injection attack. Since this value represents a table name, using a parameterized query (the common method used to avoid injection attacks) is not an option.

Next, this code verifies that the table does not already exist in the data dictionary. Once that is done, a new connection is created using the data dictionary administrative account. This connection is then used to call CREATE TABLE to create the table, and then to call GRANT for each non-default group returned in the system.usergroups table.

Note: The administrative user name and passwords are represented by string literals in this code segment. This was done for convenience, but in a real application you would either ask for this information from the user, or you would obfuscate this information so that it could not be retrieved.

Changing a User Password

A user can change the password on their own connection, if you permit this. In most cases, only when every user has a distinct user name would you expose this functionality in a client application. When multiple users share a user name, this operation is usually reserved for an application administrator.

The following event handler, associated with the Change Password button (shown in Figure 21-1), demonstrates how you can permit a user to change their password from a client application:

```
void changePasswordBtn_actionPerformed(ActionEvent e) {
  String userName;
  String oldPass;
  String newPass1;
  String newPass2;
  try {
    ResultSet rs = stmt.executeQuery("SELECT USER() as " +
     "Name FROM system.iota");
    rs.first();
    userName = rs.getString("Name");
    oldPass = JOptionPane.showInputDialog(this,
      "Enter your current password");
    if (oldPass.equals("")) {
```

```
      return;
    }
    try {
      Connection tempcon =
        DriverManager.getConnection(
        "jdbc:extendedsystems:advantage://server:6262/" +
        "share/adsbook/demodictionary.add;user=" +
        userName +";password=" + oldPass);
    }
    catch (Exception e1) {
      JOptionPane.showMessageDialog(this,
          "Invalid password. Cannot change password");
      return;
    }
    //Check for semicolon hack
    newPass1 = JOptionPane.showInputDialog(this,
      "Enter your new password");
    if (! (newPass1.indexOf(";") == -1)) {
      JOptionPane.showMessageDialog(this,
        "Password may not contain a semicolon");
      return;
    }
    newPass2 = JOptionPane.showInputDialog(this,
      "Confirm your new password");
    if (!newPass1.equals(newPass2)) {
      JOptionPane.showMessageDialog(this,
          "Passwords did not match. " + "
            Cannot change password");
      return;
    }
    stmt.executeUpdate("EXECUTE PROCEDURE "
      +"sp_ModifyUserProperty('"+
      userName + "', 'USER_PASSWORD', '" + newPass1 + "')");
    JOptionPane.showMessageDialog(this,
      "Password successfully changed. " +
        "New password will be valid next time you connect");
  } catch (Exception e1) {
    System.out.println(e1.getMessage());
  }
}
```

A number of interesting tricks are used in this code. First, the user name is obtained by requesting the USER scalar function from the system.iota table. USER returns the user name on the connection through which the query is executed. Next, the user is asked for their current password, and the user name and password are used to attempt a new connection, which, if successful, confirms that the user is valid.

Finally, the user is asked for their new password twice (for confirmation). If all is well, the sp_ModifyUserProperty stored procedure is called to change the user's password. As the final dialog box displayed by this event handler indicates, this password will be valid once the user terminates all connections on this user account.

*Note: If you run this code, and change the password of the adsuser account, you should use the Advantage Data Architect to change the password back to **password**. Otherwise, you will not be able to run this project again, since the password is hard-coded into the connection string.*