Chapter 4 JDBC Tutorial

- A Simple JDBC Application
- Prepared Statement to Improve Performance
- Transactions with JDBC
- Positioned Cursor Updates

Every program has unique requirements for database access; some need to *browse* a database, selecting a set of rows in order to allow users to move forward and backwards through the set; other applications need to perform a series of updates to a set of related tables that must be treated as a complete, *atomic* transaction, where multiple rows in the update must be treated as a single row.

Some applications contain a fixed set of selection criteria for data, while others require that parameters be provided at runtime. There are yet other applications that may know nothing about the capabilities of the database to which they are connected, so they must discover the capabilities of the database. Demonstrating a single application with JDBC cannot cover the full spectrum of functionality we reference here; a series of JDBC examples is required.

Java applications are now used primarily as applets; this more likely than not represents the most common usage of JDBC applications for the near future. But it is reasonable to expect that over time Java, with its array of features, will be accepted as a general-purpose language. Once it has been accepted, Java with JDBC will be used for a variety of general-purpose applications such as CGI programming, reports, and data entry programs.

This tutorial demonstrates the use of JDBC first in a series of simple applications, then in an applet and CGI application. The simple applications demonstrate the basics of JDBC usage: loading a database driver, creating a `Connection` object, creating a `Statement` object, executing a SQL statement with the `Statement` object and returning a `ResultSet`, and retrieving rows of data using the `ResultSet` object. Database access with JDBC will always represent some variation of these calls and additional calls as needed.

Code examples are also used to demonstrate JDBC usage with applets. This represents a variation on the simple code example; with applets, JDBC methods are usually called during button events to retrieve and display data to the applet window

A very common application currently used with World Wide Web HTML pages is the CGI application. The CGI example in this chapter uses JDBC to retrieve data from a database; it demonstrates a CGI application that receives a set of parameters, parses the parameters, and returns data formatted as an HTML page.

One of the limitations of the current implementation of JDBC is that a `ResultSet` can only be reviewed in serial order—the cursor cannot move backwards. An example demonstrated in this chapter provides a solution that allows rows to be retrieved in any order.

The overall goal of the tutorial section is to demonstrate the use of JDBC to program simple to moderately difficult database access. Complete examples are used to provide a clear understanding of the context of the application.

Tutorials are provided for the following topics:

- A Simple JDBC Application
- Use of the Prepared Statement
- Positioned Cursor Update
- Transaction Modes
- Java Applet
- Metadata Usage
- ResultSet Array

These examples are be explained in more detail in the following sections.

A Simple JDBC Application

This simple tutorial demonstrates the use of JDBC to create a `Connection` object and connect to the database; create a `Statement` object and execute a SQL statement using the `Statement` object; retrieve the results of the `Statement` in a `ResultSet` object; and to display the data in the `ResultSet`.

This example uses a class and a series of methods to

1. Create a database table
2. Insert data into the table
3. Select data from the table
4. Update rows in the table
5. Delete rows from the table

This list of database activities represents a broad spectrum of database functions; most database access programs are required to perform some or all of these functions.

Two types of SQL statements are demonstrated in this tutorial: Data Definition Language (DDL) statements and Data Manipulation Language (DML) statements. The DDL statements are used to create a database table and an index; for JDBC purposes, these statements are update statements executed with the `executeUpdate` method because they do not return data. They do however, return an integer value for the number of rows updated.

Use of the Prepared Statement

Once stored for a statement, this process of parsing and optimizing need not be repeated as long as the structure and database objects in the statement do not change. Since the overhead of parsing and optimizing a statement can be avoided during each execution of a SQL statement, a prepared statement is more efficient than regular execution of a SQL statement.

A JDBC prepared statement allows parameters to be identified within a SQL statement. The parameters are usually limited to those values that vary from execution to execution of the statement.

The prepared statement examples presented here demonstrate the use of a prepared statement to improve performance and allow parameter substitution.

Positioned Cursor Update

In many databases, you can create a cursor to maintain a pointer to a specific row in a table. This pointer or position indicates where the current row pointer is located. When the application needs to update the table being read using the cursor, it uses the cursor to update the record at the current record position; this is known as a *positioned cursor update.*

The syntax for the positioned cursor update, if supported by the target database, is usually a SQL `select` statement clause, which identifies the `select` statement as a statement to be used to create a cursor. Once the statement with the "`for update of`" clause has been declared, the cursor name is retrieved to create the update statement. The update statement SQL string includes the "`where current of`" clause followed by the cursor name.

Transaction Modes

Database transaction modes enable varying degrees of transaction integrity to be used during program execution. An application can switch from a mode where uncommitted records can be read and records updated by the application can be read by other users, to a mode where only committed records can be read by an application and no records that have been updated by an application can be read by other users. This use of granularity in transactions allows for better performance and increased concurrency when an application does not need to limit it (such as a report). But more limited concurrency may be necessary when an application needs to update several tables within a transaction and commit the rows to the database as a transaction.

The transaction example demonstrates the use of transaction modes by creating a database connection and then setting the isolation mode for the database connection through the appropriate `Connection` method. The JDBC API does not provide an explicit "begin work" statement. Using the "`commit work`" statement, all current database transactions from the session are sent to the database when the statement is executed.

This example executes the `commit Connection` method to commit the current updates to the database. A series of statements is then executed followed by another commit method invocation to commit the transaction to the database. Should the transaction fail due to some error, the `catch` code block contains a `rollback` method call to roll back the database to a current state. One of the current shortcomings of the JDBC interface is that it does not provide a means of scrolling through a `ResultSet` both forward and backward; this capability is known as *scroll cursors*. This feature is useful for a database browse application for which the user must enter selection criteria and then move backwards and forwards through the returned set of rows

The solution to this problem is to store the `ResultSet` in a Java `Vector` object. The `Vector` has the ability to grow dynamically and provides the ability to address a specific element. The result set array example demonstrates this capability.

Java Applet

The Java applet currently represents one of the principal uses of the Java language. A Java applet can be downloaded off the Internet and run through a browser. This capability has been a large part of the reason for the incredible popularity of the Java language.

A Java applet that can access a database is a powerful programming tool. This application is platform-independent and, when placed at a single location, can be distributed to multiple client computers by simply being downloaded as a Java applet through a link in the HTML page.

But a Java applet run through a browser is currently subject to certain security restrictions depending on the browser being used. For instance, a Java applet that has been downloaded cannot access any local files on the client machine. An application that wants to create a Microsoft Access table and insert rows into the table would fail as a downloaded applet if the Microsoft Access database builds files on the local machine.

The example shown here uses a Microsoft Access database that resides on the local machine. It runs successfully using the Sun `appletviewer` application where security is relaxed. It does not run using the more restrictive Netscape browser.

This example will first display an input form to the application window. Using the buttons available in this window, the user can browse the data available in the database. Search criteria can be entered and then used to retrieve rows from the database. Users can optionally move forward or backwards through the ResultSet by pressing buttons in the application window.

CGI Application

In today's world of World Wide Web/Internet application programming, CGI applications are ubiquitous. While use of JDBC in applets can eliminate the need for many of these CGI programs, security restrictions and performance improvements could still make CGI programming a viable alternative. And Java, as a flexible general-purpose language, could fill this role.

If it is desirable to have the applet or a HTML page connect to a database on a server other than the Internet server, there are a number of good reasons why you would not want to expose that machine to the Internet and would prefer to have the HTTP server process and manage the connection.

In order to connect to this machine, a third-tier application is needed as a middle tier between the client applet and the database server. A CGI application is a viable approach to programming this third tier.

Such a CGI application can receive a request, retrieve the data, and then format the data for return as an HTML page. The CGI tutorial application demonstrates a Java program that could provide output for such a CGI application.

Metadata Usage

There is a rich supply of metadata methods available in JDBC. An application can use these methods to discover information about the database to which it is connected—a task that could be a requirement for a Java applet that needs to connect to multiple databases. These examples demonstrate the use of many of the metadata functions available in JDBC.

ResultSet Array Example

One of the limitations of the current release of JDBC is that result sets can only be retrieved in a serial fashion. The `ResultSet` methods only retrieve the next row; the previous row cannot be retrieved. Using a technique that stores retrieved rows in an internal list (a Java `Vector`), data can be retrieved for the current row, the previous row, and for a specific row in the result set. This technique is demonstrated in the `ResultsSet` array example and the three-tiered application example.

Basic JDBC Programming

This chapter presents the basic steps involved in creating JDBC programs. The first example in this chapter demonstrates the basic set of calls required to use JDBC with Java. These steps are:

1. Load driver
2. Create connection
3. Create statement
4. Execute statement and return `ResultSet` or result count
5. Iterate `ResultSet` if returned

The use of JDBC usually involves some combination of these calls in addition to other calls to metadata or transaction control methods. The calls listed here must be made in sequence—you must have a `Connection` object before a `Statement` object can be created, and you must have a Statement object before a SQL statement can be executed.

Results are returned in a `ResultSet`, the JDBC equivalent of a cursor. The JDBC `ResultSet` provides methods for iterating the results and retrieving individual columns. Specific methods are used to retrieve specific data types. In the event an update is executed, an integer result count is returned.

The `ResultSet` retrieved contains, as the name implies, the *set of results* retrieved by the query. These results may be iterated, but only sequentially; there is no capability to move backwards through the result set or to move a specific set of positions. A work-around for this limitation is demonstrated later in this section.

Note

The design of JDBC has kept methods and their arguments simple. To reduce the number of parameters to be passed to methods, additional methods were added to span the functionality needed. So, instead of designing a method with three parameters, one that would indicate the call type and two others that may or may not be needed depending on the call type, JDBC developers would create three separate methods.

To discover some basic information about the `ResultSet`, a `ResultSet` metadata object must be obtained. This metadata object will provide information such as the number of columns in the `ResultSet`, the data type of the columns, and the size and precision of the column. As some of the examples in this chapter demonstrate, it is possible to convert the basic data types from the `ResultSet` to a string and display or manipulate the data in that format.

Note

If you know and are familiar with the database being used, then metadata information probably won't have to be retrieved. In situations where this information is not known, then the database metadata methods are available. Any application that can possibly connect to databases from different database vendors potentially needs metadata information. Such an application might be a general-purpose database query tool that could attach to either an Informix, Oracle, or Sybase database using JDBC drivers. This application would need to discover the database to which it was connected, the version of the database product, and potentially the specific capabilities supported in that version. All of this information is supplied by database metadata methods.

To discover information about the database or the result set, a metadata object can be instantiated using a `DataBaseMetaData` object or a `ResultSetMetaData` object. These objects provide information on the database, data types supported, or the number of columns retrieved and their data types. It is not uncommon to retrieve some metadata information about the database or the result set as is demonstrated in the examples provided here.

Basic JDBC Steps

The following sections outline the basic steps necessary to create and manage a database connection using JDBC. A specific set of methods must be invoked each time a database connection is made and data is retrieved.

Load Driver

The first step in using JDBC is to load the JDBC-ODBC bridge driver. This is usually accomplished using the `forName` static method of the `Class` object (which is part of the base Java system). The call is made as follows:

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

When this call is made, the Java system searches for the class requested and loads the driver. A class descriptor is returned by this method, but because it is not needed, it is ignored.

Create Connection

The loading of the JDBC database driver does not connect to the database; it merely creates an *environment* in the program where this can be done. Before any database-specific SQL statements can be executed, a *connection* must be established to the database. This is accomplished through a call to the `DriverManager getConnection` method to find a specific driver that can create a connection to the URL requested.

The `DriverManager` searches through registered drivers until one is found that can process the database URL that was specified. If a driver cannot be found, an exception is thrown and code execution will not continue for that method. Code that follows this statement can therefore assert that no exception was thrown and a connection has been successfully established. The call is made as follows:

```
String url     = "jdbc:odbc:msaccessdb";
Connection con = DriverManager.getConnection ( url,
"", "");
```

In this example, the `getConnection` method is invoked with a `String` containing the URL for the database and two additional `String` parameters, one for the user name and one for the user password.

Note

The familiar universal resource locator (URL) is used to supply the naming system for the database resource to be loaded. The format of the URL name is:

```
jdbc:subprotocol:subname
```

where *subprotocol* indicates the access method used in addition to JDBC and the *subname* is a name that has significance for the *subprotocol* being used.

In this case, the JDBC-ODBC bridge is being used and ODBC is the subprotocol, the protocol being used as a *bridge* to provide database connectivity. The subname in this case is the *data source* name for the ODBC connection. In this example, the data source name is `msaccessdb`, a local client Microsoft Access database. The specifics of the database name and location are mapped through the ODBC driver facilities provided.

Create Statement

In order to interact with the database, SQL statements must be executed. This requires that a Statement object be created to manage the SQL statements. This is accomplished with a call to the Connection class createStatement method as follows:

```
Statement stmt = con.createStatement( );
```

This call creates a Statement object using the established database connection. The Statement class provides methods for executing SQL statements and retrieving the results from the statement execution. Note that result sets (or cursors) are not part of the Statement class but are represented through a separate class, the ResultSet class.

Execute SQL Statement and Return ResultSet

The SQL Statement object does not have a specific SQL statement associated with it (unlike the PreparedStatement superclass, which does). The SQL statement to execute is determined when the call to executeQuery is made, as follows:

```
String qs = "select * from orders";
ResultSet rs = stmt.executeQuery( qs );
```

This call sends the query to the database and returns the results of the query as a ResultSet. Should an error be generated during the execution of the query, an exception is generated and caught using the catch code block. Successful execution of the executeQuery moves control to the next line of code following the statement, which in this example begins iterating the query results.

Iterate ResultSet

The ResultSet represents the collection of results from the query. The ResultSet class contains methods that can be used to iterate through these results in a serial fashion. First, you must make a call to the next method in order to position the pointer (or cursor) before the first element of the result set, as follows:

```
boolean more = rs.next();
```

The call to the `next` method returns a `boolean` value. The `boolean` value of *true* indicates that the call was successful and the pointer is positioned, thus there is data to retrieve. A `boolean` value of *false* indicates that the call was unsuccessful and there are no rows to retrieve. Because it is not an error to execute a SQL select statement that returns no rows, this first call to the `next` method reveals whether or not the query returned any rows—a value of *false* would indicate no rows have been retrieved.

Next, a `while` loop is executed to step through the results in the `ResultSet`. The loop control is the boolean variable `more` returned by the first call to the `next` method. As long as this value is *true*, the loop continues to execute.

Within the loop, the value of the first column of the result set is displayed and the `next` method is called to position the pointer to the next row. If the `next` method returns false, then the loop does not continue execution and control is passed to the statement after the end of the `while` loop, as follows:

```
           while ( more ) {
            System.out.println( "Col1: " +
rs.getInt( "col1" ) );
           more = rs.next();
          }
```

The complete code for the simple select program is shown in Program 4.1.

Program 4.1 Select1.java

```
    import java.sql.*;
    import java.io.*;

    class Select1 {

         public static void main( String argv[] ) {

           try {

              Class.forName ("jdbc.odbc.JdbcOdbcDriver");
```

continued

```
     String url = "jdbc:odbc:msaccessdb";

Connection con = DriverManager.getConnection (url, "",
      "");

String qs = "select * from loadtest";
Statement stmt = con.createStatement( );
ResultSet rs = stmt.executeQuery( qs );
boolean more = rs.next();
while ( more ) {
           System.out.println( "Col1: " +
                     rs.getInt( "col1" ) );

    more = rs.next();
  }
  }

  catch (java.lang.Exception ex) {

  // Print description of the exception.
  System.out.println( "** Error on data select. ** " );
    ex.printStackTrace ();

           }
       }
    }
```

A Dynamic SQL Select Program

The previous program used a specific SQL `select` statement to retrieve rows and display a single column of data from the database table. The following example presents a more generic approach to processing a SQL `select` statement. The program accepts a single command line argument: the name of the table to query. It uses this table name to build a query for all the columns and all the rows in the specified table. The query is executed and the results are displayed to the terminal screen.

Because the query is built at runtime, the number and names of the columns are not known when the program is compiled. This information must be determined by retrieving metadata information on the `ResultSet` using the `ResultSetMetaData` object for the `ResultSet` returned by the query. This example does not deal with the problem of determining the data type of the column

The next method and data retrieval

Note that calls to the `ResultSet` `next` method do not return data. They merely position the pointer to the next row in the result set. Successive calls to the appropriate "get" method for the data types of the columns must be made to retrieve the data (for example, `getInt`, `getString`, `getNumeric`). The programmer must know the data types of the columns and call the correct method. Alternatively, if simple display of data is required and the programmer does not know the data type of the column being retrieved, each column value can be retrieved as a `String` regardless of data type, as follows:

```
System.out.println( "Col1: " + rs.getString( "col1" ) );
```

In this example, the value of column 1 is retrieved as a `String` even though the column in the database is defined as an integer. This approach obviously has its limitations with data types such as BIT and BINARY, but could be useful with some of the more simple data types.

(which is easily available with the getType ResultSetMetaData method) but simply treats each column as a Java String and displays the data in the column as returned by the getString ResultSet method. The steps used in executing this program are as follows:

1. Load driver and get database connection
2. Retrieve table name from command line argument
3. Build select statement
4. Create statement object and execute SQL statement
5. Create a ResultSetMetaData object
6. Traverse the ResultSet

Each of these steps are detailed in the following sections.

Load Driver and Get Database Connection

The database driver is loaded and the connection is made as shown in the previous example. The same ODBC data source is used for this connection, as follows:

```
Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
String url     = "jdbc:odbc:msaccessdb";
Connection con = DriverManager.getConnection (
                                     url, "", "");
```

The forName method is used to load the JDBC-ODBC bridge class. The URL string is created with reference to the ODBC MicroSoft-Access database used in the example. This string is then passed as a parameter to the getConnection method of DriverManager, which then returns the Connection object.

Retrieve Table Name from Command Line Argument

This program retrieves the table name to query as a command line argument. This code determines only whether or not an argument has been passed to the program. A String variable is declared and initialized to the value of a valid table name for the database. If an argument has been passed to the program, it is stored in a String variable named tableName as shown in the following snippet. If an argument has not been passed to the program, the variable retains the original value of the table name.

```
String tableName = "loadtest";
if ( argv.length > 0 )
     tableName = argv[0];
```

Build Select Statement

The SQL select statement is built by concatenating a select column list clause with the table name stored in the tableName variable. The code for this is as follows:

```
String qs = "select * from " + tableName;
```

No where clause is appended to the SQL select statement; the query will retrieve all rows from the database table.

Create Statement Object and Execute SQL Statement

Then the Statement object is created using the Connection object and the SQL statement is executed using the executeQuery method, as follows:

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery( qs );
```

The executeQuery method returns a ResultSet, which is then processed as shown in the following steps.

Create a ResultSetMetaData Object

A ResultSetMetaData object is then created. This is used to determine the characteristics of the ResultSet that has been retrieved. The ResultSet getMetaData method is used to retrieve this object, as follows:

```
ResultSetMetaData rsmd = rs.getMetaData();
```

The code used to retrieve and display the ResultSet follows. First an integer index variable is created and the next method is called for the ResultSet. Calling the next method positions the pointer for the ResultSet at the first result row and determines whether or not there are any rows to retrieve. The boolean return value from the next method (a Java boolean variable named more) is then used to control a while loop, as follows:

```
int n = 0;
boolean more = rs.next();
while ( more ) {
    for ( n = 1; n <= rsmd.getColumnCount(); n++ ) {
         System.out.println( "Col " + n +
                   " Name: " + rsmd.getColumnName( n ) +
                   " value: " + rs.getString( n ) );
                    }
         }
```

For each iteration of the while loop, all columns in the row are retrieved and displayed. This is accomplished using an inner for loop that iterates up to the count returned by the getColumnCount method of ResultSetMetaData. For each column value returned, a call to the ResultSetMetaData getColumnName method returns the column name. Each column value is returned as a String value using the getString method of the ResultSet class.

The complete code for the dynamic SQL select program is shown in Program 4.2.

Program 4.2 selectgen.java

```
import java.sql.*;
import java.io.*;

class SelectGen {

     public static void main( String argv[] ) {

       try {
```

continued

```
           Class.forName ("jdbc.odbc.JdbcOdbcDriver");
           String url = "jdbc:odbc:msaccessdb";
           Connection con = DriverManager.getConnection (
                                    url, "", "");

           String tableName = "loadtest";
           if ( argv.length > 0 )
              tableName = argv[0];

           String qs = "select * from " + tableName;
           Statement stmt = con.createStatement();

           ResultSet rs = stmt.executeQuery( qs );
           ResultSetMetaData rsmd = rs.getMetaData();

           int n = 0;
           boolean more = rs.next();
               while ( more ) {
                 for ( n = 1; n <=
     rsmd.getColumnCount(); n++ ) {
                 System.out.println( "Col " + n +
                                  " Name: " + 

       rsmd.getColumnName( n ) +
                                   " value: " +
                                   rs.getString( n )
       );
                      }
            }

        }

        catch (java.lang.Exception ex) {
```

continued

```
     // Print description of the exception.
     System.out.println( "** Error on data select. ** " );
     ex.printStackTrace ();
             }

   }

}
```

Prepared Statement

Each SQL query presented to the database engine must be processed before data can be retrieved or updated. The database engine must determine whether or not the SQL statement presented to it is syntactically correct, whether the database objects referenced exist in the engine, and whether the data type conversions necessary can be performed. These basic operations are known as *parsing* the SQL statement. In addition to parsing the query, the database engine must make decisions about what the best access path is to process the SQL statement. This process is known as *optimizing* the SQL statement. Both of these operations require a certain amount of overhead in the database engine. If a query is to be performed many times with the same structure, then it may be better to perform these operations once and merely substitute parameters for the portions of the query that change with each successive execution. This can be accomplished with JDBC using the `PreparedStatement` class.

The `PreparedStatement` class allows a SQL statement to be *prepared* with place-holders for the parameters. These place-holders are usually the "?" character and they can only be used to create parameters for certain portions of the SQL statement. Many databases do not allow database objects (table and column names) to be substituted with parameters.

(This does not preclude creating queries at runtime where the table names and column names are not known. This can still be accomplished by building a `String` with the query and using the `executeQuery` or `executeUpdate` method of the `Statement` class to execute the SQL statement.)

Using PreparedStatement for data retrieval offers performance improvements over queries executed with the Statement class methods. The code shown in this example was used to test this claim. A version of this program (included at the end of this section) contains the same SQL statement execution but instead of preparing the statement, the SQL statement is created using string concatenation and then is executed using the executeQuery method of the Statement class. This version of the program took 126 seconds to complete 2000 iterations. The same SQL statement executed using a prepared statement completed in 24 seconds.

The use of a PreparedStatement also provides a convenient way to define queries in a single location in the code, and then using the prepared statement (represented by a PreparedStatement object) throughout the program.

The program shown here creates and executes a prepared SQL statement in the following steps.

1. Load driver and create connection
2. Create query string with parameters and create PreparedStatement object
3. Set parameter value and execute query
4. Loop for 2000 iterations

Load Driver and Create Connection

As shown previously, the database driver is loaded and the connection to the database is made. The same ODBC data source is used for this connection, as follows:

```
Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
String url     = "jdbc:odbc:msaccessdb";
Connection con = DriverManager.getConnection (
                                      url, "", "");
```

The forName method is used to load the JDBC-ODBC bridge class. The URL string is created with reference to the ODBC MicroSoft-Access database used in the example. This string is then passed as a parameter to the getConnection method of DriverManager, which then returns the Connection object.

Create Query String with Parameters and Create PreparedStatement Object

A `String` used to hold the query is created and assigned an initial value of the SQL `select` statement with the placeholder in the `where` clause, as follows:

```
String qs = "select * from loadtest where col1 = ? ";
PreparedStatement prepStmt = con.prepareStatement( qs );
```

The `PreparedStatement` object, `prepStmt`, in combination with the `setInt` method in the `preparedStatment` class, is now used to execute the statement throughout the program.

Set Parameter Value and Execute Query

The goal of this program is to demonstrate the performance improvement that can be realized with the execution of prepared SQL statements. The starting time and ending time therefore are tracked using a series of calls to a `java.util.Date` object, as follows:

```
          Date dt = new Date();
          long seconds = dt.getTime();

          String startTime =
DateFormat.getTimeInstance().format( dt );
          System.out.println( "Start Time: " +
startTime );

          int n = 3;
          boolean result;

          prepStmt.setInt( 1, n );
          ResultSet rs = prepStmt.executeQuery();
```

The value of the prepared statement parameter must be set before the query is executed. This is accomplished using the `setInt` method to set the value of the parameter. The `setInt` method takes two arguments, an integer value indicating the position of the parameter (starting from position 1) in the query statement and an integer value to set the parameter at that position. Once the parameter is set, the executeQuery method of the PreparedStatement class is called to execute the statement and return a ResultSet representing the results of the query.

Loop for 2000 Iterations

In the next step, the result set is positioned before the start of the first set and the loop is started. In this test, data is not actually retrieved and displayed (this does not significantly affect the results). For each iteration, the previous ResultSet is closed, the PreparedStatement parameter is set to the new value using the index variable for the for loop, and the executeQuery method is called and the new ResultSet is retrieved using the same object container that was previously used.

```
          boolean more = rs.next();
          for (; n < 2000 && more ; n++ ) {

              rs.close();
              prepStmt.setInt( 1, n );
              rs   = prepStmt.executeQuery();
              more = rs.next();

          }

          Date dtEnd = new Date();
          long endSeconds = dtEnd.getTime();
          String endTime =
DateFormat.getTimeInstance().format( dtEnd );
          System.out.println( "End Time:" + endTime );

       // display elapsed time
       seconds = (endSeconds - seconds)/1000;
       System.out.println( "Elapsed time: " + seconds +
                 " seconds for " + n + " records." );
```

When the loop is complete, the ending time and the elapsed time are calculated and displayed to the terminal screen.

The complete code for this example is shown in Program 4.3.

Program 4.3 preptest2.Java

```
import java.sql.*;
import java.io.*;
import java.util.Date;

class PrepTest2 {

     public static void main( String argv[] ) {

       try {

          Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
          String url = "jdbc:odbc:msaccessdb";

          Connection con = DriverManager.getConnection (
                                  url, "", "");

          String qs = "select * from loadtest where col1 = ? ";
          PreparedStatement prepStmt =
      con.prepareStatement( qs );

          Date dt = new Date();
          long seconds = dt.getTime();

          String startTime = DateFormat.getTimeInstance()
      .format( dt );
          System.out.println( "Start Time: " + startTime
      );
          int n = 3;
          boolean result;

          prepStmt.setInt( 1, n );
          ResultSet rs = prepStmt.executeQuery();
          boolean more = rs.next();
          for (; n < 2000 && more ; n++ ) {
              rs.close();
              prepStmt.setInt( 1, n );
              rs   = prepStmt.executeQuery();
```

continued

```
               more = rs.next();

           }

           Date dtEnd = new Date();
           long endSeconds = dtEnd.getTime();
          String endTime =
       DateFormat.getTimeInstance().format( dtEnd );
          System.out.println( "End Time:" + endTime );
           // display elapsed time
           seconds = (endSeconds - seconds)/1000;
           System.out.println( "Elapsed time: " + seconds +
                            " seconds for " + n + "
       records." );

        }

        catch (java.lang.Exception ex) {

       // Print description of the exception.
       System.out.println( "** Error on data select. ** " );
       ex.printStackTrace ();

       }
    }
}
```

The following code example shows the creation and execution of a query statement to process the same number of records but uses a `Statement` object instead of a `PreparedStatement` to process the SQL statement. The query statement is created within the processing loop using the following code.

```
   String qs = "select * from loadtest where col1 = ";
...
   queryString   = qs + n;
   rs  = stmt.executeQuery( queryString );
```

Because the new value for the selection criteria cannot be related to a parameter, with each iteration of the loop the query string must be re-created and then must be executed using the executeQuery method of the Statement class. The query string has been defined as a string with the column select criteria missing. This information can be appended to the query string to complete the statement and is done for each iteration of the loop, as shown in Program 4.4.

Program 4.4 preptest1.Java

```
import java.sql.*;
import java.io.*;
import java.util.Date;
import java.tsxt.DateFormat;

class PrepTest1 {

   public static void main( String argv[] ) {

     try {

        Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
        String url = "jdbc:odbc:msaccessdb";
        Connection con = DriverManager.getConnection (
                                url, "", "");
        Statement stmt = con.createStatement();

        Date dt = new Date();
        long seconds = dt.getTime();

       String startTime = DateFormat.getTimeInstance()
      .format( dt );
       System.out.println( "Start Time: " + startTime );

        int n = 1;
        String qs = "select * from loadtest where col1 = ";
```

continued

```
         String queryString = qs + n;
         ResultSet rs = stmt.executeQuery( queryString );
         boolean more = rs.next();

         for (; n < 2000 && more ; n++ ) {
             queryString   = qs + n;
             rs  = stmt.executeQuery( queryString );
             more = rs.next();
         }

         Date dtEnd = new Date();
         long endSeconds = dtEnd.getTime();
          String endTime =
       DateFormat.getTimeInstance().format( dtEnd );
          System.out.println(  End Time:  + endTime );

         // display elapsed time
         seconds = (endSeconds - seconds)/1000;
         System.out.println( "Elapsed time: " + seconds +
                         " seconds for " + n + " records."
       );

        }

     catch (java.lang.Exception ex) {

    // Print description of the exception.
    System.out.println( "** Error on data insert. ** " );
    ex.printStackTrace ();
    }
}
}
```

It is not uncommon for an application to read data with a cursor and then update rows selectively based on information gathered during the data retrieval process. It is convenient and more efficient simply to update “the current row” of the cursor rather than to create selection criteria and execute another SQL statement to search for and then update the record. The additional statement execution could require an index read and possibly additional data retrieval.

The positioned cursor update (or *update cursor*) provides functionality that eliminates the need to query for an update of a current record. This capability is supported in JDBC provided the database being used supports it. This example performs the following steps:

1. Load database driver and create connection
2. Create `DatabaseMetaData` object and test for positioned update functionality
3. Execute select query
4. Get cursor name and execute update statement
5. Review results

Load Database Driver and Create Connection

The JDBC-ODBC bridge driver is loaded as in the previous steps. The only difference in this case is that the database driver loaded is the Informix database driver. This driver is needed because the Microsoft Access database used in the previous examples does not support positioned update as of this writing.

```
Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
String url = "jdbc:odbc:informix5";
Connection con = DriverManager.getConnection (
                       url,      // database URL
                       "usera",  // user name
                       "xxxxx"); // user password
```

The call to create the Connection object includes values for the user name and password. These values are required by the Informix database being used.

Create DatabaseMetaData Object and Test for Positioned Update Functionality

Once the connection is established, the program tests for the ability to perform positioned updates. This is accomplished using the DatabaseMetaData object for the database connection.

```
   // need a database that supports positioned updates
   DatabaseMetaData dmd = con.getMetaData();
   if ( dmd.supportsPositionedUpdate() == false )
{
        System.out.println(
       "Positioned update is not supported by this
database."  );
            System.exit( -1 );
        }
```

The DatabaseMetaData object is created using the getMetaData method of the Connection object. The DatabaseMetaData class contains a supportsPositionedUpdate method that returns true if positioned updates are supported and returns false if they are not. In the previous code snippet, if the supportsPostionedUpdate method returns false then an error message is printed to the terminal screen and the program terminates.

Execute Select Query

Two Statement objects are used to perform the database operations: one Statement to retrieve the data and set the cursor position and the other to perform the update. The statement executed to retrieve the data is created and executed as follows:

```
Statement stmt1 = con.createStatement();
ResultSet rs = stmt1.executeQuery( "select " +
               " * from loadtest where col1 = 5" +
               " for update " );
```

This statement is executed using a `select` statement that ends with the clause “for update.” This indicates to the database engine that the cursor may be used later to perform an update.

Get Cursor Name and Execute Update Statement

The common SQL syntax for performing a positioned update is

```
update <table_name>
set    <column_list> = <value_list>
where current of <cursor_name>
```

The cursor name is needed to perform a positioned update. This name is obtained using the `getCursorName` method of the `ResultSet` class as shown:

```
String cursName = rs.getCursorName();
System.out.println( "cursor name is " + cursName );

Statement stmt2 = con.createStatement();

// update stmt2 at col1 = 5
int result = stmt2.executeUpdate(
      "update loadtest set col2 =  1000  " +
      " where current of " + cursName );
```

A second `Statement` is created and the cursor name is used to create the update statement executed with the `executeUpdate` method of the `Statement` class. The cursor name is appended to the clause “where current of” to identify a cursor for the positioned update statement.

Review Results

This example then executes another statement that retrieves data from the updated row. This data is then displayed to the terminal screen to validate that the update has taken place, as shown in the following code:

```
// retrieve row to view updated value
rs = stmt1.executeQuery( "select * from loadtest " +
       " where col1 = 5 " );

 rs.next();
 System.out.println( " col1 = " + rs.getInt( 1 ) +
                     " col2 = " + rs.getInt( 2 ) );
```

The complete code for this example is shown in Program 4.5.

Program 4.5 posupd.java

```
import java.sql.*;
import java.io.*;

class PosUpd {

     public static void main( String argv[] ) {

       try {

          Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
          String url = "jdbc:odbc:informix5";
          Connection con = DriverManager.getConnection (
                         url, "usera", "xxxxx");

          // need a database that supports positioned updates
          DatabaseMetaData dmd = con.getMetaData();

          if ( dmd.supportsPositionedUpdate() == false ) {
              System.out.println(
          "Positioned update is not supported by this database." );
              System.exit( -1 );
          }

          Statement stmt1 = con.createStatement();
```

continued

```
            ResultSet rs = stmt1.executeQuery( "select " +
                           " * from loadtest where col1 = 5" +
                           " for update " );
            rs.next(); // look at the first row (col1=5)

            String cursName = rs.getCursorName();
            System.out.println( "cursor name is " +
        cursName );

            Statement stmt2 = con.createStatement();

            // update stmt2 at col1 = 5
            int result = stmt2.executeUpdate(

                "update loadtest set col2 =  1000  " +
                " where current of " + cursName );

            // retrieve row to view updated value
            rs = stmt1.executeQuery( "select * from
        loadtest " +
               " where col1 = 5 " );

            rs.next();
            System.out.println( " col1 = " + rs.getInt( 1 ) +
                                " col2 = " + rs.getInt( 2 ) );

         }

         catch (java.lang.Exception ex) {

        // Print description of the exception.
        System.out.println( "** Error on data select. ** " );
        ex.printStackTrace ();

        }
     }
  }
```

Transaction Modes

Transactions provide the capability to treat a series of SQL update statements as a single statement; if any statement fails, the entire set of updates is removed from the database. If a database supports transactions, JDBC provides the facilities to use these transactions.

With JDBC, if a database supports transactions and transaction logging is on, then every statement is treated as though a transaction were open. There is no explicit “begin work” to indicate the start of a transaction because the database is always in a transaction. A `commit` method is available in the `Connection` class to commit all current work to the database and begin a new transaction. This effectively executes a “begin work” against the database.

A JDBC connection begins with the database in *auto-commit* mode. This means that every SQL statement executed is treated as an individual transaction; no statements will be grouped together as transactions. This mode must be changed using the `setAutoCommit` method of the `Connection` class. The following steps are involved in the creation of the transaction modes example.

1. Load driver and create connection
2. Set the auto-commit mode
3. Create statement and execute DDL and DML
4. Commit work
5. Create prepared statement and execute updates
6. Rollback work and examine results
7. `catch` code block

These steps are detailed in the sections that follow.

Load Driver and Create Connection

The JDBC-ODBC bridge driver is loaded first. The database driver loaded is the Informix database driver because support for transactions is needed in this example.

```
Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
String url = "jdbc:odbc:informix5";
Connection con = DriverManager.getConnection (
                        url,
                        "usera",
                        "xxxxx");
```

The call to create the `Connection` object includes values for the user name and password. These values were required by the Informix database being used.

Set the Auto-Commit Mode

When the JDBC auto-commit mode is set to true, each SQL statement is executed as a *singleton* transaction; if it completes successfully, there is an implied commit to the database. This mode would preclude the grouping of a set of SQL statements as one single, atomic transaction. Setting the auto-commit mode to *false* disables the auto-commit feature and allows a group of SQL statements to be grouped as a transaction.

```
  // will turn off the default auto-commit mode so that statements
  // can be grouped as transactions.
  con.setAutoCommit( false );
```

Create Statement and Execute DDL and DML

A statement object is required to execute a series of SQL statements to update the database. DDL statements are then executed to create a database table and create an index on the database table.

```
          Statement stmt = con.createStatement();


  int result = stmt.executeUpdate(
      "create table transtest( col1 int, col2 int, col3 char(10)
)" );
  result =    stmt.executeUpdate(
      "create index idx1 on transtest( col1 ) " );
```

Commit Work

If an error occurs during the execution of any of the previous SQL statements, a `SQLException` is thrown and caught with the `catch` code block in the method. This code block executes a SQL *rollback*, which rolls back or removes from the database the results of the execution of the statements shown in the previous method. If code execution has arrived at the following line, then no fatal exception has been thrown and the data can be committed to the database. This can be accomplished using the `commit` method of the `Connection` class.

```
con.commit();
```

Note that in some databases, executing a `commit` or `rollback` would close open database statements, requiring database objects to be re-opened after these operations.

Create Prepared Statement and Execute Updates

To demonstrate multiple updates and transactions, a series of updates will be performed as a single transaction. A `preparedStatement` object is created using the `prepareStatement` method of the `Connection` object. This returns a statement with a single parameter which is substituted before the statement is executed as shown below.

```
...
int n = 0;
PreparedStatement prepStmt = con.prepareStatement(
  " insert into transtest values ( ?, 1,  XXXXXXX  ) " );

for ( n = 1; n < 20; n++ ) {
    prepStmt.setInt( 1, n );
    prepStmt.executeUpdate();
}
```

Within the `for` loop, the single statement parameter is set and the prepared statement is executed using the `executeUpdate` statement. This loop will be executed and the database update performed 20 times. This entire set of updates will represent a single transaction.

Rollback Work and Display Results

To demonstrate the effect of a rollback work statement, the `rollback` method of the `Connection` object is executed. This rolls back the work since the last commit. This means that the database table and the index remain in the database after the rollback method has been executed because these statements were executed before the commit work method had been called.

```
con.rollback();

// validate that rollback succeeded. There should be
//no data in the table
Statement stmt1= con.createStatement();
ResultSet rs = stmt1.executeQuery( "select * from transtest" );
boolean more = rs.next();
if ( more == false )
   System.out.println( "Data was rolled back ");
```

After the rollback work has been executed, a new statement is created and executed to examine the data that remains. If no data is found, this indicates that the table is still there, but there is no data in the table—an indication that the rollback was successful.

catch Code Block

This section of code will be executed if an `SQLException` has been thrown.

This indicates that an error has occurred and all of the statements in the group should be rolled back. This rollback is performed as follows:

```
 catch (SQLException  ex) {

 // Print description of the exception.
 System.out.println( "** Error on database update. Rolling back
... ** " );
 con.rollback();
 ex.printStackTrace ();

 }
```

Program 4.6 provides the complete code for the transaction mode example.

Program 4.6 TransData.Java

```
import java.sql.*;
import java.io.*;
class TransData {

  public static void main( String argv[] ) {

    try {

       Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
       String url = "jdbc:odbc:msaccessdb";
       Connection con = DriverManager.getConnection (
                                  url, "", "");

       // will turn off the default auto-commit mode so that
      statements
       // can be grouped as transactons.
       con.setAutoCommit( false );

       Statement stmt = con.createStatement();

       int result =
         stmt.executeUpdate(
              "create table transtest( col1 int, col2 int,
      col3 char(10) )" );

       result =
         stmt.executeUpdate(
           "create index idx1 on transtest( col1 ) " );

       con.commit();
       int n = 0;
       PreparedStatement prepStmt = con.prepareStatement(
```

continued

```
           " insert into transtest values ( ?, 1, 'XXXXXXX' ) " );

        for ( n = 1; n < 20; n++ ) {
            prepStmt.setInt( 1, n );
            prepStmt.executeUpdate();
        }

        con.rollback();

// validate that rollback succeeded.
// There should be no data in the table
        Statement stmt1= con.createStatement();
        ResultSet rs = stmt1.executeQuery( "select * from
       transtest" );
        boolean more = rs.next();
        if ( more == false )
           System.out.println( "Data was rolled back ");

     }
     catch (SQLException  ex) {

    // Print description of the exception.
    System.out.println( "** Error on database update.
       Rolling back ... ** " );
   con.rollback();
   ex.printStackTrace ();

    }
   }
}
```

CGI Application

With the prevalence of the World Wide Web, CGI applications are commonplace. Though currently these are written primarily in C or C++, Java presents an attractive alternative to these languages for the creation of these applications. The code in this section provides an example of a simple CGI application written in Java.

The purpose of the this CGI program is to retrieve the records from the customer's table where the last name is *like* the parameter passed into the CGI program. The CGI application first receives the command line arguments, the CGI token. This token is parsed and used as a parameter in a SQL statement to be executed. The results of the executed statement are formatted as an HTML page and displayed to the terminal screen. The following steps are used in this application:

1. Load driver manager and create connection
2. Create prepared statement with parameter
3. Parse CGI arguments
4. Set parameters and execute query
5. Retrieve results and HTML output

These steps are discussed in more detail in the following sections.

Load Driver Manager and Create Connection

The driver manager is loaded as in the previous examples and the connection is created with the Microsoft Access database. The code for this is as follows:

```
Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
String url = "jdbc:odbc:msaccessdb";
Connection con = DriverManager.getConnection (
                              url, "", "");
```

Create Prepared Statement with Parameter

A `PreparedStatement` object is then created using a SQL `select` statement that includes a parameter for the filter statement. This parameter is used to identify the list of customer table records that will be displayed in the HTML page. The value for this parameter is supplied by the CGI parameters passed to the program.

```
    PreparedStatement stmt  = con.prepareStatement(
                     " select * from customers " +
                     " where lastname like ? " );
```

Parse CGI Arguments

The CGI parameters are passed to the program using a “+” to separate the arguments. These arguments must be parsed and the parameter values retrieved from the string passed to the program.

First the command line array is checked to determine whether or not any arguments have been passed to the program. If no arguments have been passed, the program will exit.

```
...
  // parse the CGI arguments
  if ( argv.length == 0 ) {
     System.out.println( "Invalid Parameters. Exiting ... " );
     System.exit( -1 );
  }

  StringTokenizer Params = new StringTokenizer(
                             argv[0], delim );
  Vector vParams = new Vector();
  String s = null;

  while ( Params.hasMoreTokens() ) {

         s = Params.nextToken();
         vParams.addElement( s );
  }
...
```

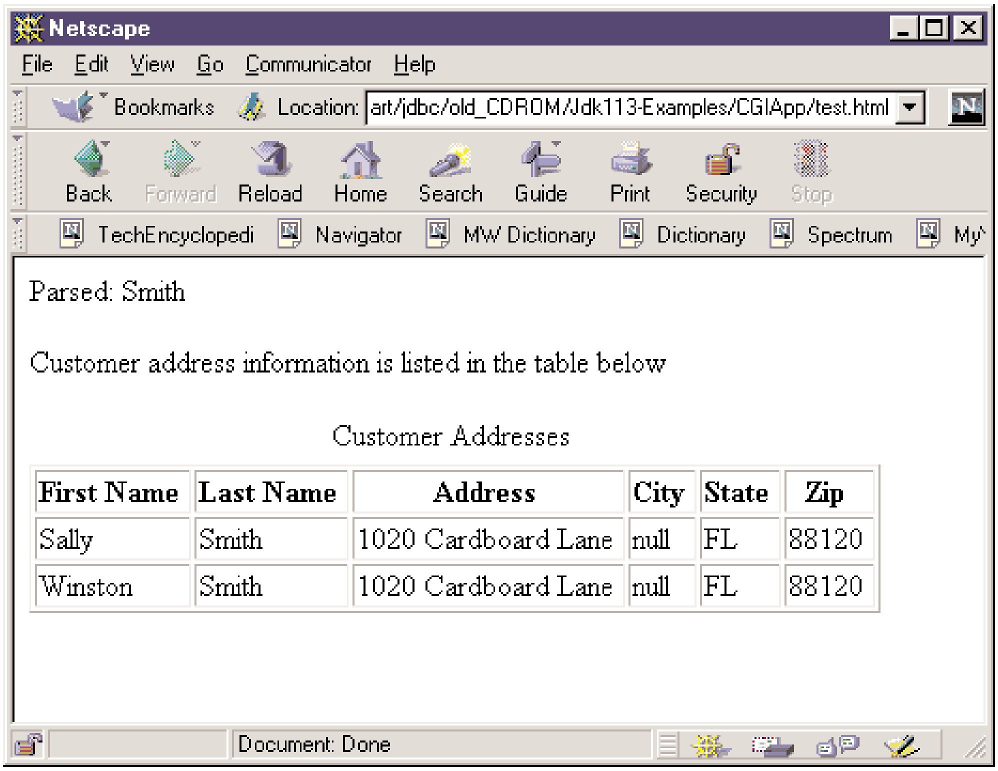
Next, a `StringTokenizer` object is created using the array of strings passed on the command line and specifying the delimiter string (previously set to the “+” character) to be used to parse the string. A `Vector` object is also created to store the parameters passed on the command line. A `while` loop is then executed to retrieve each of the parameter values passed. As each of these values is retrieved, it is added to the Vector used to store the parameter values. (In this example, only one parameter value is passed.)

Set Parameters and Execute Query

The parameter values then are used as parameters for the query. This is accomplished by retrieving the parameter value from the Vector object used to store the values and using this string to set the first parameter in the PreparedStatement containing the query.

```
// Arg1 is the last name


stmt.setString( 1, vParams.elementAt( 0 ).toString() );
ResultSet rs = stmt.executeQuery();
```


Netscape
File Edit View Go Communicator Help
Bookmarks Location: art/jdbc/old_CDROM/Jdk113-Examples/CGIApp/test.html
Back Forward Reload Home Search Guide Print Security Stop
TechEncyclopedi Navigator MW Dictionary Dictionary Spectrum My
Document: Done

Parsed: Smith

Customer address information is listed in the table below

Customer Addresses

First Name	Last Name	Address	City	State	Zip
Sally	Smith	1020 Cardboard Lane	null	FL	88120
Winston	Smith	1020 Cardboard Lane	null	FL	88120

Figure 4.1: Output of CGI demonstration application

The results of the query are then retrieved in a `ResultSet`. If no results have been retrieved, as indicated by the `boolean` value returned from the `next ResultSet` method call, then the program displays an error message and exits. If program execution continues, then values have been found and will be displayed using formatting commands for the HTML pages. These formatting commands display the page as an HTML table, as shown in Figure 4.1.

```
  ResultSetMetaData rsmd = rs.getMetaData();

  boolean more = rs.next();
  if ( !more ) {
     System.out.println( "Error - no rows retrieved" );
     System.exit( -1 );
  }

  // HTML page header
  System.out.println( "</ul>" );
  System.out.println(
"<p> Customer address information is listed in the table below
</p>" );

  // Table header
  System.out.println( "<table border > " );
  System.out.println( "<caption>Customer Addresses </caption> " );
  System.out.println( "<th> First Name </th>" );
  System.out.println( "<th> Last Name </th>" );
  System.out.println( "<th> Address </th> " );
  System.out.println( "<th> City </th> " );
  System.out.println( "<th> State </th> " );
  System.out.println( "<th> Zip </th> " );

  // display the table rows
  while ( more ) {
        System.out.println( "<tr> " );
        for ( n = 1; n <= rsmd.getColumnCount(); n++ )
               System.out.println( "<td > "  +
                            rs.getString( n ) +
                             " </td> " );
        System.out.println( "</tr>" );
        more = rs.next();
   }
  System.out.println( "</table> " );

}
```

A `Statement` object is used to determine the number of columns in the retrieved `ResultSet`. Each column of the retrieved row is placed in the table, the result being a HTML table with rows of data for each row returned from the database.

The complete code for this example is displayed in Program 4.7.

Program 4.7 cgiapp.java

```
import java.sql.*;
import java.util.StringTokenizer;
import java.util.Vector;
import java.io.*;

class cgiApp {

     static String delim = "+";

public static void main( String argv[] ) {

   int n = 0;

  try {

     Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
     String url = "jdbc:odbc:msaccessdb";
     Connection con = DriverManager.getConnection (
                                  url, "", "");

     PreparedStatement stmt  = con.prepareStatement(
                      " select * from customers " +
```

continued

```
                 " where lastname like ? " );


   // parse the CGI arguments
   if ( argv.length == 0 ) {
      System.out.println( "Invalid Parameters. Exiting ...
   " );
      System.exit( -1 );
   }

   StringTokenizer Params = new StringTokenizer( argv[0],
    delim );
   Vector vParams = new Vector();
   String s = null;

   while ( Params.hasMoreTokens() ) {

          s = Params.nextToken();
          vParams.addElement( s );
   }

   // Arg1 is the last name
   s = vParams.elementAt(0).toString();

   stmt.setString( 1, vParams.elementAt( 0 ).toString() );
   ResultSet rs = stmt.executeQuery();

   ResultSetMetaData rsmd = rs.getMetaData();

   boolean more = rs.next();
   if ( !more ) {
      System.out.println( "Error - no rows retrieved" );
      System.exit( -1 );
   }

   // HTML page header
   System.out.println( "</ul>" );
```

continued

```
        System.out.println( "<p> Customer address information is
          listed in the table below </p>" );

        // Table header
        System.out.println( "<table border > " );
        System.out.println( "<caption>Customer Addresses
          </caption> " );
        System.out.println( "<th> First Name </th>" );
        System.out.println( "<th> Last Name </th>" );
        System.out.println( "<th> Address </th> " );
        System.out.println( "<th> City </th> " );
        System.out.println( "<th> State </th> " );
        System.out.println( "<th> Zip </th> " );

        while ( more ) {
              System.out.println( "<tr> " );
              for ( n = 1; n <= rsmd.getColumnCount(); n++ )
                  System.out.println( "<td > "  +
                                  rs.getString( n ) +
                                   " </td> " );
              System.out.println( "</tr>" );
              more = rs.next();
        }
        System.out.println( "</table> " );

      }

      catch ( java.lang.Exception ex ) {
          ex.printStackTrace();
      }
    }
  }
```

Metadata Access

The JDBC interface provides access to a rich supply of information about the current database or a `ResultSet`. While many users never need to access this information, there is most likely some small subset that will be useful to most users. For instance, the

ResultSetMetaData class provides information on the number of columns retrieved in a ResultSet. It is very likely that generic routines reading a ResultSet will want to make use of this information rather than hard-coding the column count each time the routine is used.

The following example demonstrates the use of metadata methods for evaluating an unknown query at runtime. This example enables the user to enter a query and then processes the query, using metadata methods to determine the number and type of columns, and making a rudimentary attempt to format the data based on the data type. This program uses the following steps:

1. Retrieve query from the command line
2. Load driver and create connection
3. Create statement and execute the query
4. Retrieve the ResultSet and determine the number of columns
5. Execute formatting routine
6. Iterate results displaying formatted data

These steps are explained in more detail in the following sections.

Retrieve Query from the Command Line

The first step is to retrieve the query as a String from the command line. This is accomplished by setting the queryString string to the value of the first element of the argument string array (argv). If this value is null, the program displays an error message and aborts. This string is then used to execute the query.

```
// default query is NULL
String queryString = null;

// default data source name
String url   = "jdbc:odbc:msaccessdb";

//  rst argument is the query to execute
if ( argv.length > 0 )
   queryString = argv[0];
```

```
      // if no query, must abort
      if ( queryString == null ) {
         System.out.println(
            "Must enter a query as a  parameter. Aborting. " );
         System.exit(-1);
     }
```

Load Driver and Create Connection

As in the previous examples, the DriverManager must be loaded and the Connection object must be created. The url string is used to connect to a local Microsoft Access database using the database URL.

```
Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con = DriverManager.getConnection (
                  url, "", "");
```

Create Statement and Execute the Query

The Statement object is then created and the query string received on the command line is executed. The results of the query execution are returned as a ResultSet. This ResultSet then is used to retrieve and process the results.

```
// Create statement
Statement stmt = con.createStatement( );

// Execute the query
ResultSet rs = stmt.executeQuery( queryString );
```

Retrieve the ResultSet and Determine the Number of Columns

A ResultSetMeta object is created from the ResultSet returned by the query statement execution. One of the more common uses of a ResultSetMetaData object is the retrieval of the number of columns returned by the ResultSet using the getColumnCount method as shown in the following code.

```
    // Determine the nature of the Results
    ResultSetMetaData md = rs.getMetaData();

    // display the results
    int numCols = md.getColumnCount();
```

Execute Formatting Routine

The `ResultSetMetaData` object is used to determine the nature of the data returned by the query. The `formatOutputString` routine is used to interpret and format the data. It receives three parameters: the `ResultSetMetaData` object, the `ResultSet` object, and the column index. The `OutputString` is the string that is returned by the method, and the `colTypeNameString` is the string used to store the data type name of the column data type:

```
  // Formatting routine
    static String formatOutputString( ResultSetMetaData rsmd,
                           ResultSet   rs,
                           int        colIndex ) {
    String OutputString = null;
    String colTypeNameString = null;

    try {

      int colType  = rsmd.getColumnType( colIndex );

      colTypeNameString = typeNameString( colType );
      if ( colTypeNameString.equals( "UNKNOWN" ) ||
        colTypeNameString.equals( "OTHER"  ) )
        colTypeNameString = rsmd.getColumnTypeName( colIndex );

      Object obj   = formattedValue( rs, rsmd, colIndex,
colType );
     if ( obj == null )
        return ( " ** NULL ** " );

     OutputString = rsmd.getColumnLabel( colIndex ) +
                    " Data Type is " +
                    colTypeNameString +
                    " ; value is " +  obj.toString();
    }
```

The getColumnType method of the ResultSetMetaData class is called to retrieve the column type of the ResultSet column being formatted (referenced by the colIndex parameter).

This method then calls the formattedValue method to format the data in the column based on the column data type. This method returns an object that is tested for a NULL value. If the object is NULL, then a string indicating a NULL value is returned. If the object is not null, a String is created with the column label as returned by the getColumnLabel method of the ResultMetaData object, the data type name as stored in the colTypeNameString variable, and the value of the object as returned by the Object class toString method. This String is returned by the method as shown in the return clause shown following the catch code block in the following code.

```
  catch ( SQLException ex ) {


             System.out.println ("\n*** SQLException
caught ***\n");


             while (ex != null) {
                System.out.println ("SQLState: " +
                        ex.getSQLState ());
                System.out.println ("Message:  " +
                        ex.getMessage ());
                System.out.println ("Vendor:   " +
                        ex.getErrorCode ());
                        ex = ex.getNextException ();
                System.out.println ("");
            }
        }

return( OutputString );

}
```

The typeNameString method evaluates the integer data type value returned by the ResultSetMetaData getColType method and simply maps the integer value to a character string name. This character string name then is displayed with the column data to indicate the column data type.

```
  // return the type name as a string
  static String typeNameString( int Type ) {

    switch ( Type ) {

  case ( Types.BIGINT ):        return ( "BIGINT" );
  case ( Types.BINARY ):        return ( "BINARY" );
  case ( Types.BIT ):           return ( "BIT" );
  case ( Types.CHAR ):          return ( "CHAR" );
  case ( Types.INTEGER ):       return ( "INTEGER" );
  case ( Types.DATE ):          return ( "DATE" );
  case ( Types.DECIMAL ):       return ( "DECIMAL" );
  case ( Types.FLOAT ):         return ( "FLOAT" );
  case ( Types.LONGVARBINARY ): return ( "LONGVARBINARY" );
  case ( Types.LONGVARCHAR ):   return ( "LONGVARCHAR" );
  case ( Types.OTHER ):         return ( "OTHER" );

    }

return  ( "UNKNOWN" );

  }
```

The formattedValue method demonstrates the process of formatting column data based on data type. The method receives a ResultSet object, a ResultSetMetaData object, a column index, and a data type for the column. The method returns an Object reference.

The method evaluates the data type being passed into the method. Based on the data type, the correct ResultSet "get" method is called to retrieve the data. The correct data type object is identified as the return value for each "get" method, but when the object is returned from the method, it is cast as an Object reference. This allows the return value to be managed in a generic way in the calling method.

There is no specific effort to format the data in this example, though that could easily be managed in the appropriate `case` clause of the `switch` statement shown in the following code. In some cases, the method does map several data types to a single Java data type, but there is no effort made to drastically change the format of the specific data in the columns.

Each `case` clause in this `switch` statement returns an `Object` reference for the specific data type returned. Should control fall through the `switch` statement, a return statement returns the object reference for the `ResultSet` column (`getObject`).

```
  static Object formattedValue( ResultSet rs,
                                ResultSetMetaData rsmd,
                                int colIndex,
                                int Type ) {


    Object generalObj = null;


         try {


    switch ( Type ) {


       case ( Types.BIGINT ):
          Long longObj = new Long( rs.getLong(colIndex ) );
              return ( (Object) longObj );
       case ( Types.BIT ):
           Boolean booleanObj = new Boolean( rs.getBoolean(
colIndex ) );
            return ( (Object) booleanObj );
       case ( Types.CHAR ):
          String stringObj = new String( rs.getString( colIndex ) );
            return ( (Object) stringObj );
       case ( Types.INTEGER ):
            Integer integerObj = new Integer( rs.getInt( colIndex )
);
           return ( (Object) integerObj );
       case ( Types.DATE ):
           Date dateObj = rs.getDate( colIndex );
           return ( (Object) dateObj );
       case ( Types.DECIMAL ):
```

```
       case ( Types.FLOAT ):
           Numeric numericObj = rs.getNumeric( colIndex,
rsmd.getScale( colIndex ) );
           return ( (Object) numericObj );

       case ( Types.BINARY ):
       case ( Types.LONGVARBINARY ) :
       case ( Types.LONGVARCHAR ) :
       case ( Types.OTHER ) :
           return ( rs.getObject( colIndex ) );

  }
  // get the object handle
  generalObj = rs.getObject( colIndex );
 }
```

Iterate Results Displaying Formatted Data

The `ResultSet` is iterated first by positioning the pointer before the first element using the `next` method, and then moving through the `ResultSet` using a `while` loop. For each row in the `ResultSet`, the row count is displayed and an inner loop displays the output of the `formatOutputString` method.

```
   // Display data, fetching until end of the result set
   boolean more = rs.next();
   int rowCount = 0;
   while (more) {

          rowCount++;
         System.out.println( "*** row " + rowCount + " *** " );

          // Loop through each column, getting the
          // column data and displaying

          for (n=1; n<=numCols; n++)
                // display formatted data
                System.out.println( formatOutputString(
md,rs, n ));
                System.out.println("");
                more = rs.next();
          }
  }
```

The complete code for this example is shown in Program 4.8.

Program 4.8 MetaDataExample1.Java

```
import java.net.URL;
import java.sql.*;

class MetaDataExample1 {

     public static void main( String argv[] ) {
       short n = 0;

       try {

        // default query is NULL
        String queryString = null;

        // default data source name
        String url   = "jdbc:odbc:msaccessdb";

        //  rst argument is the query to execute
        if ( argv.length > 0 )
           queryString = argv[0];

        // if no query, must abort
        if ( queryString == null ) {

           System.out.println(
                 "Must enter a query as a  parameter.
      Aborting. " );
           System.exit(-1);
        }

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

continued

```
    Connection con = DriverManager.getConnection (
                      url, "", "");

    // Create statement
    Statement stmt = con.createStatement( );

    // Execute the query
    ResultSet rs = stmt.executeQuery( queryString );

    // Determine the nature of the Results
    ResultSetMetaData md = rs.getMetaData();

    // display the results
    int numCols = md.getColumnCount();

    System.out.println("");

    // Display data, fetching until end of the result set
    boolean more = rs.next();
    int rowCount = 0;
    while (more) {

      rowCount++;
     System.out.println( "*** row " + rowCount + " ***
  " );

           // Loop through each column, getting the
           // column data and displaying

           for (n=1; n<=numCols; n++)
                 // display formatted data
                  System.out.println(
  formatOutputString( md,rs, n ));
                 System.out.println("");
                 more = rs.next();
           }
 }
```

continued

```
     catch ( SQLException ex ) {

                System.out.println (
                   "\n*** SQLException caught ***\n");

                while (ex != null) {
                  System.out.println ("SQLState: " +
                    ex.getSQLState ());
                  System.out.println ("Message:  " +
                    ex.getMessage ());
                  System.out.println ("Vendor:   " +
                    ex.getErrorCode ());
                  ex = ex.getNextException ();
                  System.out.println ("");
                }
        }
        catch (java.lang.Exception ex) {

                // Got some other type of exception.
  Dump it.

                ex.printStackTrace ();
        }
}


 // Formatting routine
 static String formatOutputString( ResultSetMetaData rsmd,
                       ResultSet         rs,
                       int               colIndex ) {
   String OutputString = null;
   String colTypeNameString = null;

   try {

     int colType  = rsmd.getColumnType( colIndex );
```

continued

```
      colTypeNameString = typeNameString( colType );
      if ( colTypeNameString.equals( "UNKNOWN" ) ||
           colTypeNameString.equals( "OTHER"  ) )
           colTypeNameString = rsmd.getColumnTypeName(
    colIndex );




      Object obj   = formattedValue( rs, rsmd,
    colIndex, colType );
      if ( obj == null )
         return ( " ** NULL ** " );

      OutputString = rsmd.getColumnLabel( colIndex ) + " Data
    Type is " +
                  colTypeNameString +
                  " ; value is " +  obj.toString();



     }



   catch ( SQLException ex ) {

                      System.out.println ("\n*** SQLEx-
    ception caught ***\n");

                while (ex != null) {
                    System.out.println ("SQLState: " +
                      ex.getSQLState ());
                    System.out.println ("Message:  " +
                      ex.getMessage ());
                    System.out.println ("Vendor:   " +
                      ex.getErrorCode ());
                      ex = ex.getNextException ();
                    System.out.println ("");
```

continued

```
                    }
             }

    return( OutputString );

    }



  // return the type name as a string

static String typeNameString( int Type ) {

  switch ( Type ) {
    case ( Types.BIGINT ):     return ( "BIGINT" );
    case ( Types.BINARY ):     return ( "BINARY" );
    case ( Types.BIT ):        return ( "BIT" );
    case ( Types.CHAR ):       return ( "CHAR" );
    case ( Types.INTEGER ):     return ( "INTEGER" );
    case ( Types.DATE ):       return ( "DATE" );
    case ( Types.DECIMAL ):    return ( "DECIMAL" );
    case ( Types.FLOAT ) :     return ( "FLOAT" );
    case ( Types.LONGVARBINARY ) : return (
       "LONGVARBINARY" );
    case ( Types.LONGVARCHAR ) : return (
       "LONGVARCHAR" );
    case ( Types.OTHER ) :     return ( "OTHER" );

    }



   return  ( "UNKNOWN" );

  }

  static Object formattedValue( ResultSet rs,
                           ResultSetMetaData rsmd,
```

continued

```
                             int colIndex,
                             int Type ) {

     Object generalObj = null;

          try {
     switch ( Type ) {
          case ( Types.BIGINT ):
            Long longObj = new Long( rs.getLong(colIndex ) );
               return ( (Object) longObj );
          case ( Types.BIT ):
             Boolean booleanObj = new Boolean(
       rs.getBoolean( colIndex ) );
             return ( (Object) booleanObj );
          case ( Types.CHAR ):
             String stringObj = new String( rs.getString( colIndex
       ) );
             return ( (Object) stringObj );
          case ( Types.INTEGER ):
            Integer integerObj = new Integer( rs.getInt(
       colIndex ) );
              return ( (Object) integerObj );
          case ( Types.DATE ):
              Date dateObj = rs.getDate( colIndex );
              return ( (Object) dateObj );
          case ( Types.DECIMAL ):
          case ( Types.FLOAT ):
            Numeric numericObj = rs.getNumeric( colIndex,
       rsmd.getScale( colIndex ) );
              return ( (Object) numericObj );

          case ( Types.BINARY ):
          case ( Types.LONGVARBINARY ) :
          case ( Types.LONGVARCHAR ) :
          case ( Types.OTHER ) :
              return ( rs.getObject( colIndex ) );
```

continued

```
     }
     // get the object handle
     generalObj = rs.getObject( colIndex );

   }
        catch ( SQLException ex ) {

  System.out.println ("\n*** SQLException caught ***\n");

                     while (ex != null) {
                     System.out.println ("SQLState: " +
                       ex.getSQLState ());
                     System.out.println ("Message:  " +
                       ex.getMessage ());
                     System.out.println ("Vendor:   " +
                       ex.getErrorCode ());
                     ex = ex.getNextException ();
                     System.out.println ("");
                }
         }


    // just return the object referernce
    return  ( generalObj );

   }
   }
```

Scrolling ResultSet Array

One of the limitations of the `ResultSet` is that *scroll* cursors are not supported. To overcome this limitation, the Java/JDBC programmer can make use of a small set of methods that provide this capability. These minor code changes provide the ability to move forward or backward through the data set, or to move to a specific row.

The following steps are taken in this program:

1. Declare RSArray object
2. Load DriverManager and connection
3. Create Statement and execute
4. Iterate ResultSet adding to ResultSetArray buffer
5. Display results

These steps are described in more detail in the following sections.

Declare RSArray Object

An RSArray object is declared to hold the ResultSet elements returned by the Statement object. This object contains the methods to store any ResultSet elements. The RSArray class contains a number of methods that will take any object reference passed (preferably a ResultSet object, but that is not required). These objects are stored in a Vector object; one for the ResultSet object pointer and the other for the columns. (The RSArray class is described later in this chapter.)

```
        static RSArray rsBuff = new RSArray();
```

Load DriverManager and Connection

The DriverManager must be loaded and a Connection established. This code establishes a Microsoft Access database connection with a local database using the JDBC-ODBC bridge.

```
      Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
String url = "jdbc:odbc:msaccessdb";
Connection con = DriverManager.getConnection ( url, "", "" );
```

Create Statement and Execute

Next, the Statement object is created and executed using a query that retrieves all columns and all rows for the loadtest table.

```
Statement stmt = con.createStatement();
ResultSet rs   = stmt.executeQuery( " select * from loadtest" );
```

Iterate ResultSet Adding to ResultSetArray Buffer

The ResultSet retrieved by executing the statement is then read in a serial fashion. Each row retrieved is added to the ResultSetArray object. At the end of the while loop, the number of records loaded (which is limited to 50 in this example) is stored in the rowsLoaded integer variable.

```
     while ( more && n++ < 50 ) {

            rsBuff.addElement( rs );

            more = rs.next();
    }
    int rowsLoaded = n;
```

Display Results

The results are then displayed in a serial fashion by using a RSArray method that displays a specific Vector element. A for loop is executed for the number of rows that have been loaded into the RSArray object. For each iteration of the loop, the BuffelementAt method returns a Vector data type for the element index value passed into the method. This Vector is the columns Vector for the row being displayed. By looping through the number of columns in the query ResultSet (as returned by the getColumnCount method of the ResultSetMetaData) all of the columns in the row will be displayed.

The Vector, named ColumnsVector, that has been returned by the RSArray elementAt method is then traversed. For each element in the Vector, the elementAt method returns an object, and the toString method converts the Object to a String for display.

```
  System.out.println( "Processed " + n + " rows" );
  // traverse the rs buffer vector ResultsBuffer
  Vector columnsVector = null;
  for ( x = 0; x < rowsLoaded-1; x++ ) {

     // get the row
     columnsVector = (Vector) rsBuff.ElementAt( x+1 );

     // display the row contents (columns)
     for ( n = 0; n < rs.getMetaData().getColumnCount(); n++ ) {

     System.out.println( "Row " + x +  " Column: " + n + " " +
                     columnsVector.elementAt( n
).toString() );
             }
            }
       }
```

Tip

Note that because the element is retrieved as an object, it is possible to determine the data type of the object by determining the name of the class. The code to perform this function would be as follows:

```
Object obj = columns.elementAt( x );
String s = obj.getClass().getName();
```

This code retrieves the `Object` reference for the specified element and then retrieves the class of the object and then calls the `getName` method to retrieve the name of the class. Using this class name, the data type of the object can be determined and then used accordingly.

The code for the entire application is presented in Program 4.9.

Program 4.9 RSArray1.java

```
import java.sql.*;
import java.io.*;
import java.util.Vector;

class rsArray1 {

     static RSArray rsBuff = new RSArray();
     public static void main( String argv[] ) {

       try {

          Class.forName ("jdbc.odbc.JdbcOdbcDriver");
          String url = "jdbc:odbc:msaccessdb";
          Connection con = DriverManager.getConnection (
      url, "", "" );

          Statement stmt = con.createStatement();
          ResultSet rs   = stmt.executeQuery( " select *
      from loadtest" );

          int n = 0;
          int x = 1;
          ResultSetMetaData rsmd = rs.getMetaData();

          boolean more = rs.next();
          int colCount = rsmd.getColumnCount();

          while ( more && n++ < 50 ) {

              rsBuff.addElement( rs );

              more = rs.next();

         }
```

continued

```
         int rowsLoaded = n;
         System.out.println( "Processed " + n + " rows" );
         // traverse the rs buffer vector ResultsBuffer

         Vector columnsVector = null;
         for ( x = 0; x < rowsLoaded-1; x++ ) {

             // get the row
             columnsVector = (Vector) rsBuff.ElementAt( x+1
       );

             // display the rows contents (columns)
             for ( n = 0; n <
       rs.getMetaData().getColumnCount(); n++ ) {

                System.out.println( "Row " + x +  "
       Column: " + n + " " +
                        columnsVector.elementAt( n
       ).toString() );
              }
           }
      }

      catch (java.lang.Exception ex) {

     // Print description of the exception.
     System.out.println( "** Error on data select. ** " );
     ex.printStackTrace ();

     }
  }
 }
```

The RSArray Class

The `RSArray` class as used in the previous example provides a means of moving forward and backward through the `ResultSet`. The `RSArray` class is composed of the following methods.

Class Definition

The RSArray class contains two Vector objects as instance variables. The ResultsBuffer Vector object is used to hold an array of Vector objects that contain the constituent columns of each of the rows. Instance variables are used to avoid having to instantiate new Vector objects each time the methods are called. The class definition for the RSArray class is as shown in the following code.

```
class RSArray {
   // instance variables
   int index = 0;

   // a vector of result sets
   Vector ResultsBuffer = new Vector();

   // a vector of rows (results columns and data values)
   Vector columns = new Vector();
```

A series of methods are used to manipulate the internal Vector objects. These methods are used to add elements to the RSArray object, retrieve an element at a specific position in the Object, or to retrieve the next or previous element in the array. These methods are:

- AddElement
- ElementAt
- next
- previous

These methods are described in more detail in the following sections.

AddElement

The addElement method takes a single ResultSet as its parameter. Each of the columns in this ResultSet are retrieved as an Object and added to the columns Vector object used to store the data in the ResultSet columns. A for loop is used to retrieve each of the columns in the ResultSet using the getObject method.

The object containing the columns is then cloned using the Object class method clone. This cloned object is then added to the

ResultsBuffer Vector object. The elements in the columns Vector then is cleared for the next iteration.

Note

Java objects are passed by reference, so passing the original Object object would lead to problems. Cloning the object makes a new copy thus effectively passing the object by value.

```
addElement(
    void addElement( ResultSet rs ) {
    int x;

    try {

        // store the columns in a Vector
        for ( x = 1;
              x <= rs.getMetaData().getColumnCount();
              x++ )
            columns.addElement( (Object) rs.getObject( x ) );

        // store the columns Vector in the Results Vector
        ResultsBuffer.addElement( (Object) columns.clone() );
        columns.removeAllElements();

    }

    catch ( java.lang.Exception ex ) {

          ex.printStackTrace();
    }
  }
// ------------------------------
```

ElementAt Method

The ElementAt method is used to retrieve the Vector element at the index position passed into the method as a parameter. It returns the element at the index position as an Object by calling the elementAt method of the ResultBuffer. The result of the operation is returned as an Object reference.

```
  Object ElementAt( int targetIndex ) {
   Vector returnVector = null;

   try {
     returnVector = (Vector) ResultsBuffer.elementAt( targetIndex-1 );
    }

   catch ( java.lang.Exception ex ) {
      ex.printStackTrace();
   }

  return ( (Object) returnVector );

  }
```

Next

The next method retrieves the next sequential element in the RSArray. It increments the internal index element and then attempts to retrieve the element at that position.

```
 Object next() {
 index++;

 return ( ElementAt( index ) );

 }
```

Previous

The previous method retrieves the previous method in the array. It first decrements the internal index and then attempts to retrieve the previous element in the RSArray.

The complete code for this program is presented in Program 4.10.

Program 4.10 RSArrayGen.java

```
import java.sql.*;
import java.io.*;
import java.util.Vector;

class RSArray {
   // instance variables
   int index = 0;

   // a vector of result sets
   Vector ResultsBuffer = new Vector();

   // a vector of rows (results columns and data values)
   Vector columns = new Vector();

       void addElement( ResultSet rs ) {
       int x;

       try {

           // store the columns in a Vector
           for ( x = 1;
               x <= rs.getMetaData().getColumnCount();
               x++ )
            columns.addElement( (Object) rs.getObject( x ) );

           // store the columns Vector in the Results
     Vector
           ResultsBuffer.addElement( (Object)
     columns.clone() );
           columns.removeAllElements();

       }

       catch ( java.lang.Exception ex ) {

             ex.printStackTrace();
```

continued

```
       }
     }

  // ------------------------------------------

  Object ElementAt( int targetIndex ) {

   Vector returnVector = null;

   try {
     returnVector = (Vector) ResultsBuffer.elementAt(
     targetIndex-1 );

   }

   catch ( java.lang.Exception ex ) {
      ex.printStackTrace();

   }
  return ( (Object) returnVector );

  }

 Object next() {
 index++;

 return ( ElementAt( index ) );

 }

 Object previous() {
 index ;

 return ( elementAt( index ) );

 }
}
```

Summary

This chapter has presented tutorials that demonstrated both the basics of JDBC and more advanced topics. The first example covered basic database access with JDBC and demonstrated the process of creating a connection to a database and retrieving data.

The process of retrieving and processing data with the JDBC `ResultSet`, a requirement for almost all JDBC applications, was demonstrated in several code examples. The important topics covered in the chapter are as follows:

- Database metadata reveals information about the nature of the database connection.
- `ResultSet` metadata reveals information about the nature of the results returned from the database.
- Using the `PreparedStatement` class to prepare a SQL statement provides performance gains and can simplify coding.
- To overcome the JDBC limitation of unidirectional cursors, results can be stored in a `Vector` object; this `Vector` object can then be used to access data randomly.

Coming Up

One of the primary uses of Java is to create applets. The following chapter provides an uncomplicated version of JDBC usage in an applet. This applet displays an applet window, retrieves data into a `ResultSet` vector, and then allows the user to browse the data moving both forward and backward through the data.