The Concept of JDBC

There are many industrial-strength DBMSs on the market, including Oracle, DB2, Sybase, and many other popular brands. The challenge faced by Sun Microsystems in the late 1990s was to develop a way for Java developers to write high-level code that accesses all popular DBMSs. One major obstacle to overcome was a language barrier. Each DBMS defined its own low-level way to interact with programs to access data stored in its databases. This meant that low-level code written to communicate with an Oracle database might need to be rewritten to access a DB2 database.

Sun met the challenge in 1996 with the creation of the JDBC driver and the JDBC API. Both were created out of necessity because until then Java couldn't access DBMSs and therefore wasn't considered an industrial-strength programming language. The JDBC driver developed by Sun wasn't a driver at all. It was a specification that described the detailed functionality of a JDBC driver. DBMS manufacturers and third-party vendors were encouraged to build JDBC drivers that conformed to Sun's specifications. Those firms who built JDBC drivers for their products could tap into the growing Java applications market.

The specifications required a JDBC driver to be a translator that converts low-level proprietary DBMS messages to low-level messages that are understood by the JDBC API and vice versa. This meant Java programmers could use high-level JDBC interfaces that are defined in the JDBC API to write a routine that interacts with the DBMS. The JDBC interface converts the routine into low-level messages that conform to the JDBC driver specification and sends them to the JDBC driver. The JDBC driver translates the routine into low-level messages that are understood and processed by the DBMS.

JDBC drivers created by DBMS manufacturers have to:

- Open a connection between the DBMS and the J2ME application
- Translate low-level equivalents of SQL statements sent by the J2ME application into messages that can be processed by the DBMS
- Return data that conforms to the JDBC specification to the JDBC driver
- Return information, such as error messages, that conforms to the JDBC specification to the JDBC driver
- Provide transaction management routines that conform to the JDBC specification
- Close the connection between the DBMS and the J2ME application

The JDBC driver makes J2ME applications database independent, which complements Java's philosophy of platform independence. Today, JDBC drivers for nearly every commercial DBMS are available from the Sun web site (www.sun.com) or the DBMS manufacturers' web sites. Java code independence is also extended to implementation of SQL queries. SQL queries are passed from the JDBC API through the JDBC driver to the DBMS without validation. This means it is the responsibility of the DBMS to implement SQL statements contained in the query.

JDBC Driver Types

The JDBC driver specification classifies JDBC drivers into four groups. Each group is referred to as a JDBC driver type and addresses a specific need for communicating with various DBMSs. The JDBC driver types are described in the following sections.

Type 1 JDBC to ODBC Driver

Microsoft was the first company to devise a way to create DBMS-independent database programs when they created Open Database Connectivity (ODBC). ODBC is the basis from which Sun created JDBC. Both ODBC and JDBC have similar driver specifications and an API. The JDBC to ODBC driver, also called the JDBC/ODBC Bridge, is used to translate DBMS calls between the JDBC specification and the ODBC specification. The JDBC to ODBC driver receives messages from a J2ME application that conforms to the JDBC specification, as discussed previously in this chapter. Those messages are translated by the JDBC to ODBC driver into the ODBC message format, which is then translated into the message format understood by the DBMS. However, avoid using the JDBC/ODBC Bridge in a mission-critical application because the extra translation might negatively impact performance.

Type 2 Java/Native Code Driver

The Java/Native Code driver uses Java classes to generate platform-specific code—that is, code only understood by a specific DBMS. The manufacturer of the DBMS provides both the Java/Native Code driver and API classes so the J2ME application can generate the platform-specific code. The obvious disadvantage of using a Java/Native Code driver is the loss of some portability of code. The API classes for the Java/Native Code driver probably won't work with another manufacturer's DBMS.

Type 3 JDBC Driver

The Type 3 JDBC driver, also referred to as the Java Protocol, is the most commonly used JDBC driver. The Type 3 JDBC driver converts SQL queries into JDBC-formatted statements. The JDBC-formatted statements are translated into the format required by the DBMS.

Type 4 JDBC Driver

The Type 4 JDBC driver is also known as the Type 4 Database Protocol. This driver is similar to the Type 3 JDBC driver, except SQL queries are translated into the format required by the DBMS. SQL queries do not need to be converted to JDBC-formatted systems. This is the fastest way to communicate SQL queries to the DBMS.

JDBC Packages

The JDBC API is contained in two packages. The first package is called java.sql and contains core JDBC interfaces of the JDBC API. These include the JDBC interfaces that provide the basics for connecting to the DBMS and interacting with data stored in the DBMS. java.sql is part of the J2SE.

The other package that contains the JDBC API is javax.sql, which extends java.sql and is in the J2ME. Included in the javax.sql package is the JDBC interface that interacts with Java Naming and Directory Interface (JNDI) and the JDBC interface that manages connection pooling, among other advanced JDBC features.

Overview of the JDBC Process

Although each J2ME application is different, J2ME applications use a similar process for interacting with a DBMS. This process is divided into five routines: loading the JDBC driver, connecting to the DBMS, creating and executing a statement, processing data returned by the DBMS, and terminating the connection with the DBMS.

It is sometimes better to get a general understanding of how the process works before delving into the details of each routine of the process. Therefore, the next few sections provide an overview of the process and each routine. A more detailed discussion of each routine is provided later in this chapter.

Load the JDBC Driver

The JDBC driver must be loaded before the J2ME application can connect to the DBMS. The Class.forName() method is used to load the JDBC driver. Suppose a developer wants to work offline and write a J2ME application that interacts with Microsoft Access on the developer's PC. The developer must write a routine that loads the JDBC/ODBC Bridge driver called sun.jdbc.odbc.JdbcOdbcDriver. The driver is loaded by calling the Class.forName() method and passing it the name of the driver, as shown in the following code segment:

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

Connect to the DBMS

Once the driver is loaded, the J2ME application must connect to the DBMS using the DriverManager.getConnection() method. The java.sql.DriverManager class is the highest class in the java.sql hierarchy and is responsible for managing driver information. The DriverManager.getConnection() method is passed the URL of the database, along with the user ID and password if required by the DBMS. The URL is a String object that contains the driver name and the name of the database that is being accessed by the J2ME application.

The DriverManager.getConnection() returns a Connection interface that is used throughout the process to reference the database. The java.sql.Connection interface is another member of the java.sql package that manages communications between the driver and the J2ME application. It is the java.sql.Connection interface that sends statements to the DBMS for processing. Listing 10-1 illustrates the use of the DriverManager.getConnection() method to load the JDBC/ODBC Bridge and connect to the CustomerInformation database.

Listing 10-1 Open a connection with a database

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
Statement DataRequest;
private Connection Db;
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
   Db = DriverManager.getConnection(url,userID,password);
}
```

Create and Execute an SQL Statement

The next step after the JDBC driver is loaded and a connection is successfully made with a particular database managed by the DBMS is to send an SQL query to the DBMS for

processing. An SQL query consists of a series of SQL commands that direct the DBMS to do something, such as return rows of data to the J2ME application. You'll learn how to write queries in the next chapter.

The Connect.createStatement() is used to create a Statement object. The Statement object is then used to execute a query and return a ResultSet object that contains the response from the DBMS, which is usually one or more rows of information requested by the J2ME application. Typically, the query is assigned to a String object, which is passed to the Statement object's executeQuery() method, as illustrated in the next code segment. Once the ResultSet is received from the DBMS, the close() method is called to terminate the statement. Listing 10-2 retrieves all the rows and columns from the Customers table.

Listing 10-2 Retrieve all the rows from the Customers table

```
Statement DataRequest;
ResultSet Results;
try {
      String query = "SELECT * FROM Customers";
      DataRequest = Database.createStatement();
      DataRequest = Db.createStatement();
      Results = DataRequest.executeQuery (query);
      DataRequest.close();
 }
```

Process Data Returned by the DBMS

The java.sql.ResultSet object is assigned the results received from the DBMS after the query is processed. The java.sql.ResultSet object consists of methods used to interact with data that is returned by the DBMS to the J2ME application. Later in this chapter you'll learn the details of using the java.sql.ResultSet object. However, the following code is an abbreviated example that gives you a preview of a commonly used routine for extracting data returned by the DBMS. Error-catching code is purposely removed from this example in order to minimize code clutter. You'll find the completed version of this routine later in this chapter and throughout Chapter 11.

Assume for Listing 10-3 that a J2ME application requested a customer's first name and last name from a table. The result returned by the DBMS is already assigned to the ResultSet object called Results. The first time that the next() method of the ResultSet is called, the ResultSet pointer is positioned at the first row in the ResultSet and returns a boolean value. If false, this indicates that no rows are present in the ResultSet. The if statement in Listing 10-3 traps this condition and displays the "End of data" message on the screen.

A true value returned by the next() method means at least one row of data is present in the ResultSet, which causes the code to enter the do…while loop. The getString() method of the ResultSet object is used to copy the value of a specified column in the current row of the ResultSet to a String object. The getString() method is passed the name of the column in the ResultSet whose content needs to be copied, and the getString() method returns the value from the specified column.

You could also pass the number of the column to the getString() method instead of the name. However, do so only if the columns are specifically named in the SELECT statement. Otherwise you cannot be sure of the order in which the columns appear in the ResultSet, especially since the table might have been reorganized since it was created, and therefore the columns might be rearranged.

In Listing 10-3, the first column of the ResultSet contains the customer's first name, and the second column contains the customer's last name. Both of these are concatenated in this example and assigned to the printrow String object, which is displayed on the screen. This process continues until the next() method, called as the conditional argument to the while statement, returns a false, which means the pointer is at the end of the ResultSet.

Listing 10-3 Retrieving data from the ResultSet

```
ResultSet Results;
String FirstName;
String LastName;
String printrow;
boolean Records = Results.next();
if (!Records ) {
   System.out.println( "No data returned");
   return;
}
else
{
 do {
   FirstName = Results.getString (FirstName) ;
   LastName = Results.getString (LastName) ;
   printrow = FirstName + " " + LastName;
   System.out.println(printrow);
 } while ( Results.next() );
}
```

Terminate the Connection to the DBMS

The connection to the DBMS is terminated by using the close() method of the Connection object once the J2ME application is finished accessing the DBMS. The close() method throws an exception if a problem is encountered when disengaging the DBMS. You'll learn how to handle this exception later in this chapter. The following is an example of calling the close() method. Although closing the database connection automatically closes the ResultSet, it is better to close the ResultSet explicitly before closing the connection.

```
Db.close();
```

Database Connection

A J2ME application does not directly connect to a DBMS. Instead, the J2ME application connects with the JDBC driver that is associated with the DBMS. However, before this connection is made, the JDBC driver must be loaded and registered with the DriverManager as mentioned previously in this chapter. The purpose of loading and registering the JDBC driver is to bring the JDBC driver into the Java Virtual Machine (JVM). The JDBC driver is automatically registered with the DriverManager once it is loaded and is therefore available to the JVM and can be used by J2ME applications.

The Class.forName(), as illustrated in Listing 10-4, is used to load the JDBC driver. In this example, the JDBC/ODBC Bridge is the driver that is being loaded. You can replace the JDBC/ODBC Bridge with the appropriate JDBC driver for the DBMS being used in your J2EE application. The Class.forName() throws a ClassNotFoundException if an error occurs when loading the JDBC driver. Errors are trapped using the catch {} block whenever the JDBC driver is being loaded.

Listing 10-4 Load the driver and catch any exceptions that might be thrown during the process

```
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
}
catch (ClassNotFoundException error) {
   System.err.println("Unable to load the JDBC/ODBC bridge." +
error.getMessage());
   System.exit(1);
}
```

The Connection

After the JDBC driver is successfully loaded and registered, the J2ME application must connect to the database. The database must be associated with the JDBC driver, which is usually performed by either the database administrator or the system administrator. Some students who are learning JDBC programming prefer to use Microsoft Access as the DBMS because the DBMS is usually available on the student's local computer. The "Associating the JDBC/ODBC Bridge with the Database" sidebar shows how to associate the JDBC/ODBC Bridge with a Microsoft Access database.

The data source that the JDBC component will connect to is defined using the URL format. The URL consists of three parts:

- jdbc, which indicates that the JDBC protocol is to be used to read the URL
- <subprotocol>, which is the JDBC driver name
- <subname>, which is the name of the database

Associating the JDBC/ODBC Bridge with the Database

You use the ODBC Data Source Administrator to create the association between the database and the JDBC/ODBC Bridge. Here's what you need to do:

1. Select Start | Settings | Control Panel.
2. Select ODBC 32 to display the ODBC Data Source Administrator.
3. Add a new user by selecting the Add button.
4. Select the driver, and then select Finish. Use the Microsoft Access Driver if you are using Microsoft Access, otherwise select the driver for the DBMS that you are using. If you don't find the driver for your DBMS on the list, you'll need to install the driver. Contact the manufacturer of the DBMS for more information on how to obtain the driver.
5. Enter the name of the database as the Data Source name in the ODBC Microsoft Access Setup dialog box. This is the name that will be used within your Java database program to connect to the DBMS.
6. Enter a description for the data source. This is optional, but will be a reminder of the kind of data stored in the database.
7. Click the Select button. You'll be prompted to browse the directory of each hard drive connected to your computer in order to define the direct path to the database. Click OK once you locate the database, and the directory path and name of the database will be displayed in the ODBC Microsoft Access Setup dialog box.
8. Since this is your database, you can determine whether a login name and password are required to access the database.

   If so, click the Advanced button to display the Set Advanced Options dialog box. This dialog box is used to assign a login name (also referred to as a user ID) and a password to the database. Select OK. If not, skip this step.
9. When the ODBC Microsoft Access Setup dialog box appears, select OK.
10. Select OK to close the ODBC Data Source Administrator dialog box.

The connection to the database is established by using one of three getConnection() methods of the DriverManager object. The getConnection() method requests access to the database from the DBMS. It is up to the DBMS to grant or reject access. A Connection object is returned by the getConnection() method if access is granted, otherwise the getConnection() method throws an SQLException.

Sometimes the DBMS grants access to a database to anyone. In this case, the J2ME application uses the getConnection(String url) method. One parameter is passed to the method because the DBMS only needs the database identified. This is shown in Listing 10-5.

Listing 10-5 Connecting to a database using only the URL

```
String url = "jdbc:odbc:CustomerInformation";
Statement DataRequest;
Connection Db;
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
   Db = DriverManager.getConnection(url);
}
catch (ClassNotFoundException error) {
      System.err.println("Unable to load the JDBC/ODBC bridge." +
            error);
      System.exit(1);
}
catch (SQLException error) {
      System.err.println("Cannot connect to the database." + error);
      System.exit(2);
}
```

Other databases limit access to authorized users and require the J2EE to supply a user ID and password with the request to access the database. In this case, the J2ME application uses the getConnection(String url, String user, String password) method, as illustrated in Listing 10-6.

Listing 10-6 Connecting to a database using a user ID and password

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
Statement DataRequest;
Connection Db;
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
   Db = DriverManager.getConnection(url,userID,password);
 }
catch (ClassNotFoundException error) {
      System.err.println("Unable to load the JDBC/ODBC bridge." +
            error);
      System.exit(1);
}
catch (SQLException error) {
      System.err.println("Cannot connect to the database." + error);
      System.exit(2);
}
```

There might be occasions when a DBMS requires information besides a user ID and password before the DBMS grants access to the database. This additional information is referred to as "properties" and must be associated with a Properties object, which is passed to the DBMS as a getConnection() parameter. Typically, properties used to access a database are stored in a text file, the contents of which are defined by the DBMS manufacturer. The J2ME application uses a FileInputStream object to open the file and then uses the Properties object load() method to copy the properties into a Properties object. This is illustrated in Listing 10-7. Notice that the third version of the getConnection() method passes the Properties object and the URL as parameters to the getConnection() method.

Listing 10-7 Using properties to connect to the database

```
Connection Db;
Properties props = new Properties ();
try {
  FileInputStream propFileStream =
                  new fileInputStream("DBProps.txt");
  props.load(propFileStream);
}
catch(IOException err) {
   System.err.print("Error loading propFile: ");
   System.err.println (err.getMessage());
   System.exit(1);
}
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
   Db = DriverManager.getConnection(url, props);
}
catch (ClassNotFoundException error) {
      System.err.println("Unable to load the JDBC/ODBC bridge." +
            error);
      System.exit(2);
}
catch (SQLException error) {
      System.err.println("Cannot connect to the database." + error);
      System.exit(3);
}
```

Timeout

Competition to use the same database is a common occurrence and can lead to performance degradation. For example, multiple applications might attempt to access a database simultaneously. The DBMS may not respond quickly for a number of reasons, one of which might be that database connections are not available. Rather than wait for a delayed response from the DBMS, the J2ME application can set a timeout period after which the DriverManager will cease trying to connect to the database.

The public static void DriverManager.setLoginTimeout(int seconds) method can be used by the J2ME application to establish the maximum time the DriverManager waits for a response from a DBMS before timing out. Likewise, the public static int DriverManager.getLoginTimeout() method is used to retrieve from the DriverManager the maximum time the DriverManager is set to wait until it times out. The DriverManager.getLoginTimeout() returns an int that represents seconds.

Connection Pool

Connecting to a database is performed on a per-client basis. That is, each client must open its own connection to a database, and the connection cannot be shared with unrelated clients. For example, a client that needs to interact frequently with a database must either open a connection and leave the connection open during processing, or open or close and reconnect each time the client needs to access the database. Leaving a connection open might prevent another client from accessing the database should the DBMS have a limited number of connections available. Connecting and reconnecting is simply time consuming and causes performance degradation.

The release of the JDBC 2.1 Standard Extension API introduced connection pooling to address the problem. A *connection pool* is a collection of database connections that are opened once and loaded into memory so these connections can be reused without having to reconnect to the DBMS. Clients use the DataSource interface to interact with the connection pool. The connection pool itself is implemented by the application server and other J2EE-specific technologies, which hide details on how the connection pool is maintained from the client.

There are two types of connections made to the database. The first is the physical connection, which is made by the application server using PooledConnection objects. PooledConnection objects are cached and reused. The other type of connection is the logical connection. A logical connection is made by a client calling the DataSource.getConnection() method, which connects to a PooledConnection object that has already made a physical connection to the database.

Listing 10-8 illustrates how to access a connection from a connection pool. A connection pool is accessible by using the Java Naming and Directory Interface (JNDI). JNDI provides a uniform way to find and access naming and directory services independent of any specific naming or directory service.

First, a J2ME application must obtain a handle to the JNDI context, which is illustrated in the first statement in this code segment. Next, the JNDI lookup() method is called and is passed the name of the connection pool, which returns the DataSource object, called pool in this example. The getConnection() method of the DataSource object is then called, as illustrated earlier in this chapter. The getConnection() returns the logical connection to the database, which is used by the J2ME application to access the database.

The close() method of the DataSource object is called once when the J2ME application is finished accessing the database. The close() method closes the logical connection to

the database and not the physical database connection. This means that the same physical connection can be used by the next J2ME application that needs access to the database.

Listing 10-8 Connecting to a database using a connection pool

```
Context ctext = new InitialContext();
DataSource pool = (DataSource) ctext.lookup("java:comp/env/jdbc/pool");
Connection db = pool.getConnection();
// Place code to interact with the database here
db.close();
```

Statement Objects

Once a connection to the database is opened, the J2ME application creates and sends a query to access data contained in the database. The query is written using SQL, which you'll learn about in the next chapter. One of three types of Statement objects is used to execute the query. These objects are Statement, which executes a query immediately; PreparedStatement, which is used to execute a compiled query; and CallableStatement, which is used to execute store procedures.

The Statement Object

The Statement object is used whenever a J2ME application needs to execute a query immediately without first having the query compiled. The Statement object contains the executeQuery() method, which is passed the query as an argument. The query is then transmitted to the DBMS for processing. The executeQuery() method returns one ResultSet object that contains rows, columns, and metadata that represent data requested by the query. The ResultSet object also contains methods that are used to manipulate data in the ResultSet, which you'll learn about later in this chapter.

The execute() method of the Statement object is used when multiple results may be returned. A third commonly used method of the Statement object is the executeUpdate() method. The executeUpdate() method is used to execute queries that contain UPDATE and DELETE SQL statements, which change values in a row and remove a row, respectively. The executeUpdate() method returns an integer indicating the number of rows that were updated by the query. ExecuteUpdate() is used to INSERT, UPDATE, DELETE, and DDL statements.

Listing 10-9 is an enhanced version of Listing 10-2, used earlier in this chapter to illustrate how to open a database connection. The enhancements are to create a query, execute the query, and return a ResultSet. Two new objects are declared in Listing 10-9: a Statement object called DataRequest and a ResultSet object called Results. In the second try {} block, the query is assigned to the String object query. The query requests that the DBMS return all the rows from the Customers table of the CustomerInformation database.

Next, the createStatement() method of the Connection object is called to return a Statement object. The executeQuery() method of the Statement object is passed the query

and returns a ResultSet object that contains data returned by the DBMS. Finally, the close() method of the Statement object is called to close the statement.

The close() method closes all instances of the ResultSet object returned by the Statement. Failure to call the close() method might cause resources used by the Statement object to remain unavailable to other J2ME applications until the garbage routine is automatically run. Java statements used to manipulate the ResultSet are placed between the call to the executeQuery() method and the close() method.

The executeQuery() method throws an SQLException should an error occur during the processing of the query. For example, the query may contain syntax not understood by the DBMS. In this case, the DBMS returns an SQL error message that is passed along to the J2ME application by the executeQuery() method.

Listing 10-9 Using the Statement object to execute a query

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
Statement DataRequest;
ResultSet Results;
Connection Db;
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
   Db = DriverManager.getConnection(url,userID,password);
 }
catch (ClassNotFoundException error) {
      System.err.println("Unable to load the JDBC/ODBC bridge." +
             error);
      System.exit(1);
}
catch (SQLException error) {
      System.err.println("Cannot connect to the database." + error);
      System.exit(2);
}
try {
      String query = "SELECT * FROM Customers";
      DataRequest = Db.createStatement();
      Results = DataRequest.executeQuery (query);
      //Place code here to interact with the ResultSet
      DataRequest.close();
 }
catch ( SQLException error ){
     System.err.println("SQL error." + error);
     System.exit(3);
}
Db.close();
```

Listing 10-10 illustrates how to use the executeUpdate() method of the Statement object. You'll notice that Listing 10-10 is nearly identical to Listing 10-9. However, the

query updates a value in the database rather than requesting that data be returned to the J2ME application. You'll learn more about how to write queries to update values in a database in the next chapter.

Three changes are made to Listing 10-9 to illustrate the executeUpdate() method of the Statement object. First, the declaration of the ResultSet object is replaced with the declaration of an int called rowsUpdated. Next, the query is changed. The SQL UPDATE command directs the DBMS to update the Customers table of the CustomerInformation database. The value of the PAID column of the Customers table is changed to 'Y' if the value of the BALANCE column is zero.

Finally, the executeUpdate() method replaces the executeQuery() method and is passed the query. The number of rows that are updated by the query is returned to the executeUpdate() method by the DBMS and is then assigned to the rowsUpdated int. rowsUpdated can be used for many purposes within the J2ME application, such as sending a confirmation notice to the J2ME application that requested database access.

Listing 10-10 Using the execute-Update() method

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
Statement DataRequest;
Connection Db;
int rowsUpdated;
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
   Db = DriverManager.getConnection(url,userID,password);
 }
catch (ClassNotFoundException error) {
      System.err.println("Unable to load the JDBC/ODBC bridge." +
             error);
      System.exit(1);
}
catch (SQLException error) {
      System.err.println("Cannot connect to the database." + error);
      System.exit(2);
}
try {
      String query = "UPDATE Customers
                      SET PAID='Y' WHERE BALANCE = '0';
      DataRequest = Db.createStatement();
      rowsUpdated = DataRequest.executeUpdate (query);
      DataRequest.close();
 }
catch ( SQLException error ){
     System.err.println("SQL error." + error);
     System.exit(3);
}
Db.close();
```

PreparedStatement Object

An SQL query must be compiled before the DBMS processes the query. Compiling occurs after one of the Statement object's execution methods is called. Compiling a query is an overhead that is acceptable if the query is called once. However, the compiling process can become an expensive overhead if the query is executed several times by the same instance of the J2ME application during the same session.

An SQL query can be precompiled and executed by using the PreparedStatement object. In this case, the query is constructed similar to queries that were illustrated previously in the chapter. However, a question mark is used as a placeholder for a value that is inserted into the query after the query is compiled. It is this value that changes each time the query is executed.

Listing 10-11 illustrates how to use the PreparedStatement object. Listing 10-11 is very similar to Listing 10-9, in which the Statement object returned information from the Customers table. However, the query directs the DBMS to return all customer information where the customer number equals the customer number specified in the query. Notice that the query has a question mark, which is a placeholder for the value of the customer number that will be inserted into the precompiled query later in the code.

The preparedStatement() method of the Connection object is called to return the PreparedStatement object. The preparedStatement() method is passed the query that is then precompiled. The set*XXX*() method of the PreparedStatement object is used to replace the question mark with the value passed to the set*XXX*() method. There are a number of set*XXX*() methods available in the PreparedStatement object, each of which specifies the data type of the value that is being passed to the set*XXX*() method (see the section "Data Types" later in the chapter). In Listing 10-11, the setString() is used because the customer number is being passed as a string.

The set*XXX*() requires two parameters. The first parameter is an integer that identifies the position of the question mark placeholder, and the second parameter is the value that replaces the question mark placeholder. In Listing 10-11, the first question mark placeholder is replaced with the value of the second parameter.

Next, the executeQuery() method of the PreparedStatement object is called. The executeQuery() statement doesn't require a parameter because the query that is to be executed is already associated with the PreparedStatement object.

The advantage of using the PreparedStatement object is that the query is precompiled once and the set*XXX*() method called as needed to change the specified values of the query without having to recompile the query. The PreparedStatement object also has an execute() method and an executeUpdate() method, as described in the previous section.

The precompiling is performed by the DBMS and is referred to as late binding. When the DBMS receives the request, the DBMS attempts to match the query to a previously compiled query. If found, then parameters passed to the query using the set*XXX*() methods are bound and the query is executed. If not found, then the query is compiled and retained by the DBMS for later use.

The JDBC driver passes two parameters to the DBMS. One parameter is the query, and the other is an array of late binding variables. Both binding and compiling are

performed by the DBMS. The late binding is not associated with the specific object or code block where the preparedStatement() is declared.

Listing 10-11 Using the PreparedStatement object

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
ResultSet Results;
Connection Db;
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
   Db = DriverManager.getConnection(url,userID,password);
 }
catch (ClassNotFoundException error) {
      System.err.println(
             "Unable to load the JDBC/ODBC bridge." + error);
      System.exit(1);
}
catch (SQLException error) {
      System.err.println("Cannot connect to the database." + error);
      System.exit(2);
}
try {
      String query = "SELECT * FROM Customers WHERE CustNumber = ?";
      PreparedStatement pstatement = Db.preparedStatement(query);
      pstatement.setString(1, "123");
      Results = pstatement.executeQuery ();
      //Place code here to interact with the ResultSet
      pstatement.close();
 }
catch ( SQLException error ){
     System.err.println("SQL error." + error);
     System.exit(3);
}
Db.close();
```

CallableStatement

The CallableStatement is used to call a stored procedure from within a J2ME object. A stored procedure is a block of code and is identified by a unique name. The type and style of code depend on the DBMS vendor and can be written in PL/SQL, TransactSQL, C, or another programming language. The stored procedure is executed by invoking the name of the stored procedure.

The CallableStatement object uses three types of parameters when calling a stored procedure. These parameters are IN, OUT, and INOUT. The IN parameter contains any data that needs to be passed to the stored procedure and whose value is assigned using the set*XXX*() method, as described in the previous section. The OUT parameter

contains the value returned by the stored procedures, if any. The OUT parameter must be registered using the registerOutParameter() method and then is later retrieved by the J2ME application using the get*XXX*() method. The INOUT parameter is a single parameter used for both passing information to the stored procedure and retrieving information from a stored procedure using the techniques described in the previous two paragraphs.

Listing 10-12 illustrates how to call a stored procedure and retrieve a value returned by the stored procedure. Listing 10-12 is similar to other listings used in this chapter, but has been modified slightly to call a stored procedure.

The first statement in the second try {} block creates a query that calls the stored procedure LastOrderNumber, which retrieves the most recently used order number. The stored procedure requires one parameter that is represented by a question mark placeholder. This parameter is an OUT parameter that will contain the last order number following the execution of the stored procedure.

Next, the preparedCall() method of the Connection object is called and is passed the query. This method returns a CallableStatement object, which is called cstatement. Since an OUT parameter is used by the stored procedure, the parameter must be registered using the registerOutParameter() of the CallableStatement object.

The registerOutParameter() method requires two parameters. The first parameter is an integer that represents the number of the parameter, which is 1, meaning the first parameter of the stored procedure. The second parameter to the registerOutParameter() is the data type of the value returned by the stored procedure, which is Types.VARCHAR.

The execute() method of the CallableStatement object is called next to execute the query. The execute() method doesn't require the name of the query because the query is already identified when the CallableStatement object is returned by the prepareCall() query method.

After the stored procedure is executed, the getString() method is called to return the value of the specified parameter of the stored procedure, which in this example is the last order number.

Listing 10-12 Calling a stored procedure

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
String lastOrderNumber;
Connection Db;
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
   Db = DriverManager.getConnection(url,userID,password);
 }
catch (ClassNotFoundException error) {
      System.err.println("Unable to load the JDBC/ODBC bridge." +
            error);
      System.exit(1);
}
catch (SQLException error) {
```

```
      System.err.println("Cannot connect to the database." + error);
      System.exit(2);
}
try {
      String query = "{ CALL LastOrderNumber (?)}";
      CallableStatement cstatement = Db.prepareCall(query);
      cstatement.registerOutParameter(1, Types. VARCHAR);
      cstatement.execute();
                        lastOrderNumber = cstatement.getString(1);
      cstatement.close();
 }
catch ( SQLException error ){
     System.err.println("SQL error." + error);
      System.exit(3);
}
Db.close();
```

ResultSet

As you'll remember from previous sections in this chapter, a query is used to update, delete, and retrieve information stored in a database. The executeQuery() method is used to send the query to the DBMS for processing and returns a ResultSet object that contains data requested by the query.

The ResultSet object contains methods that are used to copy data from the ResultSet into a Java collection of objects or variable(s) for further processing. Data in a ResultSet object is logically organized into a virtual table consisting of rows and columns. In addition to data, the ResultSet object also contains metadata, such as column names, column size, and column data type.

The ResultSet uses a virtual cursor to point to a row of the virtual table. A J2ME application must move the virtual cursor to each row, then use other methods of the ResultSet object to interact with the data stored in columns of that row. The virtual cursor is positioned above the first row of data when the ResultSet is returned by the executeQuery() method. This means that the virtual cursor must be moved to the first row using the next() method. The next() method returns a boolean true if the row contains data, otherwise a boolean false is returned, indicating that no more rows exist in the ResultSet.

Once the virtual cursor points to a row, the get*XXX*() method is used to copy data from the row to a collection, object, or variable. As illustrated previously in this chapter, the get*XXX*() method is data type specific. For example, the getString() method is used to copy String data from a column of the ResultSet. The data type of the get*XXX*() method must be the same data type of the column in the ResultSet.

The get*XXX*() method requires one parameter, which is an integer that represents the number of the column that contains the data. For example, getString(1) copies the data from the first column of the ResultSet.

Columns appear in the ResultSet in the order in which column names appeared in the SELECT statement in the query. Let's say a query contained the following SELECT statement:

SELECT CustomerFirstName, CustomerLastName FROM Customer

This query directs the DBMS to return two columns. The first column contains customer first names, and the second column contains customer last names. Therefore, getString(1) returns data in the customer first name column of the current row in the ResultSet.

Reading the ResultSet

Listing 10-13 illustrates a commonly used routine to read values from a ResultSet into variables that can later be further processed by the J2ME application. Listing 10-13 is based on previous code segments in this chapter.

Once a successful connection is made to the database, a query is defined in the second try {} block to retrieve the first name and last name of customers from the Customers table of the CustomerInformation database. The next() method of the ResultSet is called to move the virtual pointer to the first row in the ResultSet. If there is data in that row, the next() returns a true, which is assigned the boolean variable Records. If there isn't any data in that row, Records is assigned a false value. A false value is trapped by the if statement, where the "End of data." message is displayed and the program terminates.

A true value causes the program to enter the do…while in the third try {} block, where the getString() method is called to retrieve values in the first and second columns of the ResultSet. The values correspond to the first name and last name. These values are assigned to their corresponding String object, which is then concatenated and assigned the printrow String object and printed on the screen.

The next() method is called in the while statement to move the virtual cursor to the next row in the ResultSet and determine whether there is data in that row. If so, statements within the do…while loop are executed again. If not, the program breaks out of the loop and executes the close() statement to close the Statement object, as discussed previously in this chapter.

Listing 10-13 Reading data from the ResultSet

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
String printrow;
String FirstName;
String LastName;
Statement DataRequest;
ResultSet Results;
Connection Db;
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
```

```
   Db = DriverManager.getConnection(url,userID,password);
 }
catch (ClassNotFoundException error) {
      System.err.println("Unable to load the JDBC/ODBC bridge." +
            error);
      System.exit(1);
}
catch (SQLException error) {
      System.err.println("Cannot connect to the database." + error);
      System.exit(2);
}
try {
      String query = "SELECT FirstName,LastName FROM Customers";
      DataRequest = Db.createStatement();
      Results = DataRequest.executeQuery (query);
}
catch ( SQLException error ){
     System.err.println("SQL error." + error);
      System.exit(3);
}
boolean Records = Results.next();
if (!Records ) {
   System.out.println("No data returned");
   System.exit(4);
}
try {
  do {
     FirstName = Results.getString ( 1 ) ;
      LastName = Results.getString ( 2 ) ;
      printrow = FirstName + " " + LastName;
     System.out.println(printrow);
  } while (Results.next() );
  DataRequest.close();
}
catch (SQLException error ) {
   System.err.println("Data display error." + error);
   System.exit(5);
}
```

Scrollable ResultSet

Until the release of JDBC 2.1 API, the virtual cursor could only be moved down the ResultSet object. But today the virtual cursor can be moved backwards or even positioned at a specific row. The JDBC 2.1 API also enables a J2ME application to specify the number of rows to return from the DBMS. Six methods of the ResultSet object are used to position the virtual cursor, in addition to the next() method discussed in the previous section. These are first(), last(), previous(), absolute(), relative(), and getRow().

The first() method moves the virtual cursor to the first row in the ResultSet. Likewise, the last() method positions the virtual cursor at the last row in the ResultSet. The previous() method moves the virtual cursor to the previous row. The absolute() method positions the virtual cursor at the row number specified by the integer passed as a parameter to the absolute() method.

The relative() method moves the virtual cursor the specified number of rows contained in the parameter. The parameter is a positive or negative integer, where the sign represents the direction the virtual cursor is moved. For example, a –4 moves the virtual cursor back four rows from the current row. Likewise, a 5 moves the virtual cursor forward five rows from the current row. And the getRow() method returns an integer that represents the number of the current row in the ResultSet.

The Statement object that is created using the createStatement() of the Connection object must be set up to handle a scrollable ResultSet by passing the createStatement() method one of three constants. These constants are TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENSITIVE, and TYPE_SCROLL_SENSITIVE.

The TYPE_FORWARD_ONLY constant restricts the virtual cursor to downward movement, which is the default setting. TYPE_SCROLL_INSENSITIVE and TYPE_SCROLL_SENSITIVE constants permit the virtual cursor to move in both directions. TYPE_SCROLL_INSENSITIVE makes the ResultSet insensitive to data changes made by another J2ME application in the table whose rows are reflected in the ResultSet. The TYPE_SCROLL_SENSITIVE constant makes the ResultSet sensitive to those changes.

Listing 10-14 illustrates how to reposition the virtual cursor in the ResultSet. This listing, which is a modification of the previous code segments used as examples in this chapter, retrieves customers' first names and last names from the Customers table of the CustomerInformation database. Since Listing 10-14 moves the virtual cursor in multiple directions, the TYPE_SCROLL_INSENSITIVE constant is passed to the createStatement(). This enables the use of virtual cursor control methods in the third try {} block. Initially, the virtual cursor moves to the first row of the ResultSet and then to the last row before being positioned at the second to last row of the ResultSet.

Next, the virtual cursor is positioned in the tenth row of the ResultSet using the absolute() method. Finally, the relative() method is called twice. The first time the relative() method is called, the virtual cursor is moved back two rows from the current row, which places the virtual cursor at row eight. The relative( ) method is again called to return the virtual cursor back to its original row by moving the virtual cursor two rows forward.

If you use any of these methods and end up positioning the cursor before the first record or beyond the last record, there won't be any errors thrown.

Listing 10-14 Using a scrollable virtual cursor

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
String printrow;
String FirstName;
String LastName;
Statement DataRequest;
```

```
ResultSet Results;
Connection Db;
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
   Db = DriverManager.getConnection(url,userID,password);
 }
catch (ClassNotFoundException error) {
      System.err.println("Unable to load the JDBC/ODBC bridge." +
            error);
      System.exit(1);
}
catch (SQLException error) {
      System.err.println("Cannot connect to the database." + error);
      System.exit(2);
}
try {
      String query = "SELECT FirstName,LastName FROM Customers";
      DataRequest = Db.createStatement(TYPE_SCROLL_INSENSITIVE);
      Results = DataRequest.executeQuery (query);
}
catch ( SQLException error ){
     System.err.println("SQL error." + error);
      System.exit(3);
}
boolean Records = Results.next();
if (!Records ) {
   System.out.println("No data returned");
   System.exit(4);
}
try {
  do {
      Results.first();
      Results.last();
      Results.previous();
      Results.absolute(10);
      Results.relative(-2);
      Results.relative(2);
      FirstName = Results.getString ( 1 ) ;
      LastName = Results.getString ( 2 ) ;
      printrow = FirstName + " " + LastName;
      System.out.println(printrow);
  } while (Results.next() );
  DataRequest.close();
}
catch (SQLException error ) {
   System.err.println("Data display error." + error);
   System.exit(5);
}
```

Not All JDBC Drivers Are Scrollable

Although the JDBC API contains methods to scroll a ResultSet, some JDBC drivers may not support some or all of these features, and therefore they will not be able to return a scrollable ResultSet. Listing 10-15 can be used to test whether or not the JDBC driver in use supports a scrollable ResultSet.

Listing 10-15 Testing whether a driver supports a scrollable ResultSet

```
boolean forward, insensitive, sensitive;
DataBaseMetaData meta = Db.getMetaData();
forward = meta.supportsResultsSetType(ResultSet.TYPE_FORWARD_ONLY);
insensitive = meta.supportsResultsSetType(
                        ResultSet. TYPE_SCROLL_INSENSITIVE);
sensitive = meta.supportsResultsSetType(
                        ResultSet. TYPE_SCROLL_SENSITIVE);
System.out.println("forward: " + answer);
System.out.println("insensitive: " + insensitive);
System.out.println("sensitive: " + sensitive);
```

Specify Number of Rows to Return

When the J2ME application requests rows from the ResultSet, some rows are fetched into the driver and returned at one time. Other times, all rows requested may not be retrieved at the same time. In this case, the driver returns to the DBMS and requests another set of rows that are defined by the fetch size and then discards the current set of rows. This process continues until the J2EE retrieves all rows.

Although the Statement class has a method for setting maximum rows, the method may not be effective since the driver does not implement them. In addition, the maximum row setting is for rows in the ResultSet and not for the number of rows returned by the DBMS. For example, the maximum rows can be set to 100. The DBMS might return 500 rows, but the ResultSet object silently drops 400 of them. This means all 500 rows are still pumped over the network.

The fetch size is set by using the setFetchSize() method, which is illustrated in Listing 10-16. However, all DBMS vendors may not implement the fetch size. Consult the driver documentation to determine whether fetch size is supported. If fetch size isn't supported, the methods will compile and execute, but have no effect. Don't become overly concerned about setting the fetch size because fetch size is in the area of performance tuning, which is handled by the database administrator or the network engineer.

Listing 10-16 illustrates how to set the maximum number of rows returned by the DBMS. The second try {} block in Listing 10-16 calls the createStatement() method of the Connection object and then sets the maximum number of rows to 500 using the setFetchSize() method of the Statement object.

Listing 10-16 Setting the maximum number of rows returned in a ResultSet

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
String printrow;
String FirstName;
String LastName;
Statement DataRequest;
ResultSet Results;
Connection Db;
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
   Db = DriverManager.getConnection(url,userID,password);
 }
catch (ClassNotFoundException error) {
      System.err.println("Unable to load the JDBC/ODBC bridge." +
            error);
      System.exit(1);
}
catch (SQLException error) {
      System.err.println("Cannot connect to the database." + error);
      System.exit(2);
}
try {
      String query = "SELECT FirstName,LastName FROM Customers";
      DataRequest = Db.createStatement(TYPE_SCROLL_INSENSITIVE);
      DataRequest.setFetchSize(500);
      Results = DataRequest.executeQuery (query);
}
catch ( SQLException error ){
     System.err.println("SQL error." + error);
      System.exit(3);
}
```

Updatable ResultSet

Rows contained in the ResultSet can be updated similar to how rows in a table can be updated. This is made possible by passing the createStatement() method of the Connection object the CONCUR_UPDATABLE. Alternatively, the CONCUR_READ_ONLY constant can be passed to the createStatement() method to prevent the ResultSet from being updated.

There are three ways to update a ResultSet. These are updating values in a row, deleting a row, and inserting a new row. All of these changes are accomplished by using methods of the Statement object.

Update a Value in ResultSet

Once the executeQuery() method of the Statement object returns a ResultSet, the update*XXX*() method is used to change the value of a column in the current row of the ResultSet. The *XXX* is replaced with the data type of the column that is to be updated. The update*XXX*() method requires two parameters. The first is either the number or name of the column of the ResultSet that is being updated, and the second is the value that will replace the value in the column of the ResultSet.

A value in a column of the ResultSet can be replaced with a NULL value by using the updateNull() method. The updateNull() method requires one parameter, which is the number of the column in the current row of the ResultSet. The updateNull() doesn't accept the name of the column as a parameter.

The updateRow() method is called after all the update*XXX*() methods are called. The updateRow() method changes values in columns of the current row of the ResultSet based on the values of the update*XXX*() methods.

Listing 10-17 illustrates how to update a row in a ResultSet. In this example, customer Mary Jones was recently married and changed her last name to Smith before processing the ResultSet. The updateString() method is used to change the value of the last name column of the ResultSet to 'Smith'. The change takes effect once the updateRow() method is called, but this change only occurs in the ResultSet. The corresponding row in the table remains unchanged until an update query is run, which is discussed in the next chapter.

Listing 10-17 Updating the ResultSet

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
Statement DataRequest;
ResultSet Results;
Connection Db;
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
   Db = DriverManager.getConnection(url,userID,password);
 }
catch (ClassNotFoundException error) {
      System.err.println("Unable to load the JDBC/ODBC bridge." +
            error);
      System.exit(1);
}
catch (SQLException error) {
      System.err.println("Cannot connect to the database." + error);
      System.exit(2);
}
try {
      String query = "SELECT FirstName,LastName
                      FROM Customers
                      WHERE FirstName = 'Mary' and
                            LastName = 'Jones'";
```

```
      DataRequest = Db.createStatement(ResultSet.CONCUR_UPDATABLE);
      Results = DataRequest.executeQuery (query);
}
catch ( SQLException error ){
     System.err.println("SQL error." + error);
      System.exit(3);
}
boolean Records = Results.next();
if (!Records ) {
   System.out.println("No data returned");
   System.exit(4);
}
try {
      Results.updateString ("LastName", "Smith");
      Results.updateRow();
      DataRequest.close();
}
catch (SQLException error ) {
   System.err.println("Data display error." + error);
   System.exit(5);
}
```

Delete a Row in the ResultSet

The deleteRow() method is used to remove a row from a ResultSet. Sometimes this is advantageous when processing the ResultSet because this is a way to eliminate rows from future processing. For example, each row of a ResultSet may have to pass three tests. Those that fail to pass the first test could be deleted from the ResultSet, thereby reducing the number of rows that have to be evaluated for the second test. This also deletes the row from the underlying database.

The deleteRow() method is passed an integer that contains the number of the row to be deleted. A good practice is to use the absolute() method, described previously in the chapter, to move the virtual cursor to the row in the ResultSet that should be deleted. However, the value of that row should be examined by the program to assure it is the proper row before the deleteRow() method is called. The deleteRow() method is then passed a zero integer indicating that the current row must be deleted, as shown in the following statement:

```
Results.deleteRow(0);
```

Insert a Row in the ResultSet

Inserting a row into the ResultSet is accomplished using basically the same technique used to update the ResultSet. That is, the update*XXX*() method is used to specify the column and value that will be placed into the column of the ResultSet. You can insert one or multiple columns into the new row using the same technique.

The update*XXX*() method requires two parameters. The first parameter is either the name of the column or the number of the column of the ResultSet. The second parameter is the new value that will be placed in the column of the ResultSet. Remember that the data type of the column replaces the *XXX* in the method name.

The insertRow() method is called after the update*XXX*() methods, which causes a new row to be inserted into the ResultSet having values that reflect the parameters in the update*XXX*() methods. This also updates the underlying database.

Listing 10-18 illustrates how to insert a new row in a ResultSet. In this example, the query returns the first name and last name of all customers. The name Tom Smith is inserted into the ResultSet in the third try {} block using the updateString() method. Remember that columns are numbered based on the order that the column names appear in the SELECT statement of the query. The new row is added to the ResultSet after the insertRow() method is called.

Listing 10-18 Inserting a new row into the ResultSet

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
Statement DataRequest;
ResultSet Results;
Connection Db;
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
   Db = DriverManager.getConnection(url,userID,password);
 }
catch (ClassNotFoundException error) {
      System.err.println("Unable to load the JDBC/ODBC bridge." +
            error);
      System.exit(1);
}
catch (SQLException error) {
      System.err.println("Cannot connect to the database." + error);
      System.exit(2);
}
try {
      String query = "SELECT FirstName,LastName FROM Customers";
      DataRequest = Db.createStatement(CONCUR_UPDATABLE);
      Results = DataRequest.executeQuery (query);
}
catch ( SQLException error ){
     System.err.println("SQL error." + error);
      System.exit(3);
}
boolean Records = Results.next();
if (!Records ) {
    System.out.println("No data returned");
```

```
    System.exit(4);
}
try {
      Results.updateString (1, "Tom");
      Results.updateString (2, "Smith");
      Results.insertRow();
      DataRequest.close();
}
catch (SQLException error ) {
    System.err.println("Data display error." + error);
    System.exit(5);
}
```

Transaction Processing

A transaction may involve several tasks similar to the tasks required to complete a transaction at a supermarket. In a supermarket transaction, each item purchased must be registered, the transaction must be totaled, and the customer must tender the amount of the purchase. The transaction is successfully completed only if each task is completed successfully. If one task fails, the entire transaction fails. Previously completed tasks must be reversed if the transaction fails. For example, goods that were registered must be removed from the register and returned to the shelf.

A database transaction consists of a set of SQL statements, each of which must be successfully completed for the transaction to be completed. If one fails, SQL statements that executed successfully up to that point in the transaction must be rolled back. A database transaction isn't completed until the J2ME application calls the commit() method of the Connection object. All SQL statements executed before the call to the commit() method can be rolled back. However, once the commit() method is called, none of the SQL statements can be rolled back.

The commit() method must be called regardless of whether the SQL statement is part of a transaction or not. This means that the commit() method must be issued in the previous examples in this chapter. However, the commit() method was automatically called in these examples because the DBMS has an AutoCommit feature that is by default set to true.

If a J2ME application is processing a transaction, the AutoCommit feature must be deactivated by calling the setAutoCommit() method and passing it a false parameter. Once the transaction is completed, the setAutoCommit() method is called again, this time passing it a true parameter, which reactivates the AutoCommit feature.

Listing 10-19 illustrates how to process a transaction. The transaction in this example consists of two SQL statements, both of which update the street address of rows in the Customers table. Each SQL statement is executed separately, and then the commit() method is called. However, should either SQL statement throw an SQL exception, the

catch {} block reacts by rolling back the transaction before displaying the exception on the screen.

Listing 10-19 Executing a database transaction

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
Statement DataRequest1, DataRequest2 ;
Connection Database;
try {
      Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
      Database = DriverManager.getConnection(url,userID,password);
}
catch (ClassNotFoundException error) {
     System.err.println("Unable to load the JDBC/ODBC bridge." +
           error);
     System.exit(1);
}
catch (SQLException error) {
     System.err.println("Cannot connect to the database." + error);
     System.exit(2);
 }
try {
  Database .setAutoCommit(false)
  String query1 = "UPDATE Customers SET Street = '5 Main Street' "
      "WHERE FirstName = 'Bob'";
  String query2 = "UPDATE Customers SET Street = '10 Main Street' " +
      "WHERE FirstName = 'Tim'";
  DataRequest1= Database.createStatement();
  DataRequest2= Database.createStatement();
  DataRequest.executeUpdate (query1 );
  DataRequest.executeUpdate (query2 );
  Database.commit();
  DataRequest1.close();
  DataRequest2.close();
  Database.close();
  }
  catch(SQLException ex) {
   System.err.println("SQLException: " + ex.getMessage());
   if (con != null) {
     try {
      System.err.println("Transaction is being rolled back ");
      con.rollback();
     }
    catch(SQLException excep) {
         System.err.print("SQLException: ");
         System.err.println(excep.getMessage());
     }
   }
 }
```

Savepoints

A transaction may consist of many tasks, some of which don't need to be rolled back should the entire transaction fail. Let's say there are several tasks that occur when a new order is processed. These include updating the customer account table, inserting the order into the pending order table, and sending a customer a confirmation email. Technically all three tasks must be completed before the transaction is considered completed. Suppose the email server is down when the transaction is ready to send the customer a confirmation email. Should the entire transaction be rolled back? Probably not since it is more important that the order continue to be processed (that is, delivered). The confirmation notice can be sent once the email server is back online.

The J2ME application can control the number of tasks that are rolled back by using savepoints. A *savepoint*, introduced in JDBC 3.0, is a virtual marker that defines the task at which the rollback stops. In the previous example, the task before the email confirmation notice is sent can be designated as a savepoint. Listing 10-20 illustrates how to create a savepoint. This is the same code segment as Listing 10-19, but a savepoint is created after the execution of the first UPDATE SQL statement.

There can be many savepoints in a transaction; each is identified by a unique name. The savepoint name is then passed to the rollback() method to specify the point within the transaction where the rollback is to stop. In this example, there is one savepoint called sp1. The name "sp1" is the parameter to the rollback() method in the catch {} block. The purpose of this example is to illustrate how to set and release a savepoint and how to use the savepoint name in the rollback() method. Of course, for commercial applications, you will want more rigorous code that identifies the executeUpdate() method that threw the exception, among other error-checking routines.

The releaseSavepoint() method is called to remove the savepoint from the transaction. The name of the savepoint that is to be removed is passed to the releaseSavepoint() method.

Listing 10-20 Using savepoints in a transaction

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
Statement DataRequest1, DataRequest2 ;
Connection Database;
try {
     Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
     Database = DriverManager.getConnection(url,userID,password);
}
catch (ClassNotFoundException error) {
     System.err.println("Unable to load the JDBC/ODBC bridge." +
            error);
     System.exit(1);
}
catch (SQLException error) {
     System.err.println("Cannot connect to the database." + error);
     System.exit(2);
```

```
 }
try {
  Database .setAutoCommit(false)
  String query1 = "UPDATE Customers SET Street = '5 Main Street' " +
      "WHERE FirstName = 'Bob'";
  String query2 = "UPDATE Customers SET Street = '10 Main Street' " +
      "WHERE FirstName = 'Tim'";
  DataRequest1= Database.createStatement();
  Savepoint s1 = Database.setSavepoint ("sp1");
  DataRequest2= Database.createStatement();
  DataRequest.executeUpdate (query1);
  DataRequest.executeUpdate (query2);
  Database.commit();
  DataRequest1.close();
  DataRequest2.close();
  Database.releaseSavepoint ("sp1");
  Database.close();
}
catch ( SQLException error ){
   try {
     Database.rollback(sp1);
   }
   catch ( SQLException error ){
      System.err.println("rollback error." + error.getMessage());
      System.exit(3);
   }
    System.err.println("SQL error." + error. getMessage()););
   System.exit(4);
}
```

Batch Statements

Another way to combine SQL statements into a transaction is to batch statements together into a single transaction and then execute the entire transaction. You can do this by using the addBatch() method of the Statement object. The addBatch() method receives an SQL statement as a parameter and places the SQL statement in the batch. Once all the SQL statements that make up the transaction are included in the batch, the executeBatch() method is called to execute the entire batch at the same time. The executeBatch() method returns an int array that contains the number of SQL statements executed successfully.

The int array is displayed if a BatchUpdateException error is thrown during the execution of the batch. The batch can be cleared of SQL statements by using the clearBatch() method. The transaction must be committed using the commit() method. Make sure that setAutoCommit() is set to false before executing the batch, as discussed in the previous section.

Listing 10-21 illustrates how to batch SQL statements. In this example, two SQL statements are created, as discussed previously in this chapter. Each SQL statement is added to the batch using the addBatch() method. Once both SQL statements are added to the batch, the executeBatch() method is called to execute each of the SQL statements. The commit() method is then called to commit the changes created by the SQL statement. Until the commit() method is called, the transaction can be rolled back, as described in the previous section.

Listing 10-21 Batching SQL statements into a transaction

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
Statement DataRequest;
Connection Database;
try {
      Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
      Database = DriverManager.getConnection(url,userID,password);
}
catch (ClassNotFoundException error) {
     System.err.println("Unable to load the JDBC/ODBC bridge." +
            error);
     System.exit(1);
}
catch (SQLException error) {
     System.err.println("Cannot connect to the database." + error);
     System.exit(2);
 }
try {
  Database .setAutoCommit(false)
  String query1 = "UPDATE Customers SET Street = '5 Main Street' " +
      "WHERE FirstName = 'Bob'";
  String query2 = "UPDATE Customers SET Street = '10 Main Street' " +
      "WHERE FirstName = 'Tim'";
  DataRequest= Database.createStatement();
  DataRequest.addBatch(query1);
  DataRequest.addBatch(query2);
  int [ ] updated = DataRequest.executeBatch ();
  Database.commit();
  DataRequest1.close();
  DataRequest2.close();
  Database.close();
}
catch(BatchUpdateException error) {
   System.out.println("Batch error.");
   System.out.println("SQL State: " + error.getSQLState());
   System.out.println("Message: " + error.getMessage());
   System.out.println(Vendor: " + error.getErrorCode());
   int [ ] updated  = error.getUpdatecount();
```

```
    int count = updated.length();
    for int - i = 0; i < count; i++) {
        System.out.print (updated[i]);
    }
  SQLException sql = error;
  While (sql != null)
      {
        System.out.println("SQL error " + sql);
        sql = sql.getnextException();
      }
  try{
     DataRequest.clearBatch();
  }
  catch(BatchUpdateException error) {
      System.out.println("Unable to clear the batch: " +
             error.getMessage());
  }
}
```

Keeping ResultSet Objects Open

Whenever the commit() method is called, all ResultSet objects that were created for the transaction are closed. Sometimes a J2ME application needs to keep the ResultSet open even after the commit() method is called. You can control whether or not ResultSet objects are closed following the call to the commit() method (called "holdability") by passing one of two constants to the createStatement() method. These constants are HOLD_CURSORS_OVER_COMMIT and CLOSE_CURSORS_AT_COMMIT.

The HOLD_CURSORS_OVER_COMMIT keeps ResultSet objects open following a call to the commit() method, and CLOSE_CURSORS_AT_COMMIT closes ResultSet objects when the commit() method is called.

RowSet

The JDBC RowSet object is used to encapsulate a ResultSet for use with Enterprise JavaBeans (EJB). A RowSet object contains rows of data from a table or tables that can be used in a disconnected operation. That is, an EJB can interact with a RowSet object without having to be connected to a DBMS, which is ideal for J2ME applications that have PDA clients. You can learn about EJB by picking up a copy of *J2EE: The Complete Reference*, by Jim Keogh (McGraw-Hill/Osborne, 2002).

A row set event is generated every time the cursor is moved in a RowSet and when one or multiple columns of the RowSet change. These events are described in the RowSetEvent.

When an event occurs, the RowSet object creates an instance of a RowSetEvent object and sends the instance to all RowSetListeners that are registered with the RowSet.

A RowSetListener is a class you define that implements the RowSetListener interface. The RowSetListener class must contain three methods, each of which responds to a particular event occurring in a RowSet. These methods are as follows:

```
public void cursorMoved(RowSetEvent event)
public void rowChanged(RowSetEvent event)
public void rowSetChanged(RowSetEvent event)
```

The cursorMoved() method must contain logic that reacts to movement of the cursor within the RowSet. Typically, this method calls ResultSet.getRow() to return the current row of the ResultSet. The rowChanged() method contains logic to respond when a portion of a RowSet is modified, and the rowSetChanged() has logic to respond when the entire RowSet is modified.

Listing 10-22 illustrates how to implement a RowSetListener. First you'll need to create a class that implements the RowSetListener, which is called MyRowSetListener in this example. Next, define the three required methods. Each method is passed an instance of a RowSetEvent created by the RowSet when the event occurs.

After defining the RowSetListener class, you'll need to create an instance of that class, which is called rowsetlistner. The instance of the RowSetListener is then registered with the RowSet by calling the addRowSetListener. This example assumes that the RowSet has already been created. You can deregister a RowSetListener from a RowSet by calling the rowset.removeRowSetListener(rowsetlistner) method.

Listing 10-22 Creating a RowSet Listener, assuming the instance rowset has already been created

```
MyRowSetListener rowsetlistener = new MyRowSetListener ();
rowset.addRowSetListener (rowsetlistener);
public class MyRowSetListener implements RowSetListener
{
  public void cursorMoved(RowSetEvent event)
  {
    // do something
  }
  public void rowChanged(RowSetEvent event)
  {
    // do something
  }
  public void rowSetChanged(RowSetEvent event)
  {
    // do something
  }
}
```

Autogenerated Keys

It is common for a DBMS to automatically generate unique keys for a table as rows are inserted into the table. The getGeneratedKeys() method of the Statement object is called to return keys generated by the DBMS.

The getGeneratedKeys() returns a ResultSet object. You can use the ResultSet.getMetaData() method to retrieve metadata relating to the automatically generated key, such as the type and properties. You can learn more about retrieving metadata in the next section of this chapter.

Metadata

Metadata is data about data, as discussed in Chapter 9. A J2ME application can access metadata by using the DatabaseMetaData interface. The DatabaseMetaData interface is used to retrieve information about databases, tables, columns, and indexes, among other information about the DBMS. A J2ME application retrieves metadata about the database by calling the getMetaData() method of the Connection object. The getMetaData() method returns a DatabaseMetaData object that contains information about the database and its components.

Once the DatabaseMetaData object is obtained, an assortment of methods contained in the DatabaseMetaData object are called to retrieve specific metadata. Here are some of the more commonly used DatabaseMetaData object methods:

- **getDatabaseProductName()** Returns the product name of the database
- **getUserName()** Returns the user name
- **getURL()** Returns the URL of the database
- **getSchemas()** Returns all the schema names available in this database
- **getPrimaryKeys()** Returns primary keys
- **getProcedures()** Returns stored procedure names
- **getTables()** Returns names of tables in the database

ResultSet Metadata

Two types of metadata can be retrieved from the DBMS: metadata that describes the database (as mentioned in the previous section) and metadata that describes the ResultSet. Metadata that describes the ResultSet is retrieved by calling the getMetaData() method of the ResultSet object. This returns a ResultSetMetaData object, as illustrated in the following code statement:

```
ResultSetMetaData rm = Result.getMetaData()
```

Once the ResultSet metadata is retrieved, the J2ME application can call methods of the ResultSetMetaData object to retrieve specific kinds of metadata. The more commonly called methods are

- **getColumnCount()** Returns the number of columns contained in the ResultSet
- **getColumnName(int number)** Returns the name of the column specified by the column number
- **getColumnType(int number)** Returns the data type of the column specified by the column number

There are many other methods used to retrieve practically any information you need to know about a database and the ResultSet—many more methods than can fit in this chapter. You can obtain detailed information about each of these methods by visiting Sun's web site, java.sun.com.

Data Types

The set*XXX*() and get*XXX*() methods are used throughout this chapter to set a value of a specific data type and to retrieve a value of a specific data type. The *XXX* in the name of these methods is replaced with the name of the data type. Table 10-1 contains a list of data types and their Java equivalents. You can use this list to determine the proper data name to replace the *XXX* in the two methods.

SQL Type	Java Type
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	Boolean
TINYINT	Byte
SMALLINT	Short
INTEGER	Integer
BIGINT	Long
REAL	float

Table 10-1. *Data Types for Use with the setXXX() and getXXX() Methods*

SQL Type	Java Type
FLOAT	float
DOUBLE	double
BINARY	Byte[]
VARBINARY	Byte[]
LONGVARBINARY	byte[]
BLOB	java.sql.Blob
CLOB	java.sql.Clob
ARRAY	java.sql.Array
STRUCT	java.sql.Struct
REF	java.sql.Ref
DATALINK	java.sql.Types
DATE	java.sql.date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Table 10-1. *Data Types for Use with the* setXXX() *and* getXXX() *Methods* (continued)

Exceptions

Three kinds of exceptions are thrown by JDBC methods. These are SQLException, SQLWarning, and DataTruncation. SQLException commonly reflects an SQL syntax error in the query and is thrown by many of the methods contained in the java.sql package. Hopefully the syntax errors in your code get resolved quickly. In production, this exception is most commonly caused by connectivity issues with the database. It can also be caused by subtle coding errors like trying to access an object that's been closed. For example, you try to roll back a transaction in a catch {} clause and don't check first to see if the database connection is still valid. The getNextException() method of the SQLException object is used to return details about the SQL error or a null if the last exception was retrieved. The getErrorCode() method of the SQLException object is used to retrieve vendor-specific error codes.

The SQLWarning throws warnings received by the Connection from the DBMS. The getWarnings() method of the Connection object retrieves the warning, and the getNextWarning() method of the Connection object retrieves subsequent warnings.

Whenever data is lost due to truncation of the data value, a DataTruncation exception is thrown.