



RV Institute of Technology and Management®

RV Educational Institutions®

RV Institute of Technology and Management

(Affiliated to VTU, Belagavi)

JP Nagar 8th Phase, Bengaluru - 560076

Department of Information Science and Engineering



Course Name: Advanced Java

Course Code: BIS402

Module 5 – JDBC Objects

IV Semester

2022 Scheme

Prepared By :

Dr. Vinod Kumar M & Dr. Kirankumar K,

Associate & Assistant Professor,

Department of Information

Science and Engineering RVITM, Bengaluru -560076



The Concept of JDBC

There are many industrial-strength DBMSS commercially available in the market. These include Oracle, DB2, Sybase, and many other popular brands. The challenge Sun Microsystems, Inc. faced in the late 1990s was to develop a way for Java developers to write high-level code that accesses all popular DBMSs.

One of the major obstacles for Sun Microsystems, Inc. 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 low-level code written to communicate with an Oracle database might need to be rewritten to access a DB2 database.

Sun Microsystems, Inc. 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 wasn't considered an industrial-strength programming language since Java was unable to access DBMSs.

The JDBC driver developed by Sun Microsystems, Inc. wasn't a driver at all. It was a specification that described the detail functionality of a JDBC driver. DBMS manufacturers and third-party vendors were encouraged to build JDBC drivers that conformed to Sun Microsystems, Inc.'s specifications. Those firms that 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 converted low-level proprietary DBMS messages to low-level messages understood by the JDBC API, and vice versa.

This meant Java programmers could use high-level Java data objects defined in the JDBC API to write a routine that interacted with the DBMS. Java data objects convert the routine into low-level messages that conform to the JDBC driver specification and send 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 J2EE component.
- Translate low-level equivalents of SQL statements sent by the J2EE component 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 J2EE component.

The JDBC driver makes J2EE components database independent, which complements Java's philosophy of platform independence. Today there are JDBC drivers for nearly every commercial DBMS, and they are available from the Sun Microsystems, Inc. web site (www.sun.com) or from the DBMS manufacturer's web site.

Java code independence is also extended to implementation of the 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

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 as follows:

Type 1 JDBC-to-ODBC Driver

Microsoft was the first company to devise a way to create a DBMS-independent database program when they created the Open Database Connection (ODBC). ODBC is the basis from which Sun Microsystems, Inc. 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.



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 Drivers

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 J2EE component 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

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 Java data objects of the JDBC API. These include Java data objects 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 J2EE. Included in the `javax.sql` package are Java data objects that



interact with Java Naming and Directory Interface (JNDI) and Java data objects that manage connection pooling, among other advanced JDBC features.

A Brief Overview of the JDBC Process

Although each J2EE component is different, J2EE components use a similar process for interacting with a DBMS. This process is divided into five routines. These include:

- Loading the JDBC driver
- Connecting to the DBMS
- Creating and executing a statement
- Processing data returned by the DBMS
- 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 of this chapter provide an overview of the process and each routine. A more detailed discussion of each routine is provided later in this chapter.

Loading the JDBC Driver

The JDBC driver must be loaded before the J2EE component 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 J2EE component 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 J2EE component 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, and 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 J2EE component.

The DriverManager.getConnection() method 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 communication between the driver and the J2EE component. It is the java.sql.Connection interface that sends statements to the DBMS for processing. Below code illustrates the use of the DriverManager.getConnection() method to load the JDBC/ODBC Bridge and connect to the CustomerInformation 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 a SQL Statement:

The next step, after the JDBC driver is loaded and connection is successfully made with a particular database managed by the DBMS, is to send a SQL query to the DBMS for processing. A SQL query consists of a series of SQL commands that direct the DBMS to do something such as to return rows of data to the J2EE component.

The Connect.createStatement() method 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 J2EE component.

Typically, the query is assigned to a String object, which is passed to the Statement object's executeQuery() method, which is illustrated in the next code segment. Once the ResultSet is received from the DBMS, the close() method is



called to terminate the statement. The below code retrieves all the rows and columns 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 J2EE component.

The following code is an abbreviated example that gives you a preview of a commonly used routine used to extract data returned by the DBMS. Error-catching code is purposely removed from this example in order to minimize code clutter.

Assume for below code that a J2EE component requested customers' first names and last names 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 that if false indicates that no rows are present in the ResultSet. The if statement in the code traps this condition and displays the "End of data" message on the screen.

However, 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 passing the column number. 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 because the table might be reorganized since the table was created and therefore the columns might be rearranged.

In the code, 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

```
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());
}
```



as the conditional argument to the while statement, returns a false, which means the pointer is at the end of the ResultSet.

Terminate the Connection to the DBMS

The connection to the DBMS is terminated by using the close() method of the Connection object once the J2EE component is finished accessing the DBMS. The close() method throws an exception if a problem is encountered when disengaging the DBMS. 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 J2EE component does not directly connect to a DBMS. Instead, the J2EE component 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.

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 the JDBC driver is loaded and is therefore available to the JVM and can be used by J2EE components.

The Class.forName() method as shown below, 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

```
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);
}
```

being used in your J2EE application.



The Connection

After the JDBC driver is successfully loaded and registered, the J2EE component 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 systems 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 data source that the JDBC component will connect to is defined using the URL format. The URL consists of three parts. These are

- **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.

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 a `SQLException`. Sometimes the DBMS grants access to a database to anyone. In this case, the

```
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);
}
```



J2EE component uses the `getConnection(String url)` method. One parameter is passed to the method because the DBMS only needs the database identified.

Connecting the database using only URL

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 J2EE

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

component uses the `getConnection(String url, String user, String password)` method.

```
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);
}
```

Connecting the database using userID and Password

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 J2EE component uses a `FileInputStream` object to open the file and then uses the `Properties`

```
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);
}
```

object `load()` method to copy the properties into a `Properties` object.

TimeOut

Competition to use the same database is a common occurrence in the J2EE environment and can lead to performance degradation of a J2EE application. For example, a J2EE application that needs database access requests service from an appropriate J2EE component. In turn, the J2EE component attempts to connect to the database.

However, the DBMS may not respond quickly for a number of reasons, which might include that database connections are not available. Rather than wait for a delayed response from the DBMS, the J2EE component can set a timeout period after which the `DriverManager` will cease to attempt to connect to the database.



The public static void DriverManager.setLoginTimeout(int seconds) method can be used by the J2EE component 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() method returns an int that represents seconds.

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, and then the 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 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 that is 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 the name of the database will be displayed in the ODBC Microsoft Access Setup dialog box.
8. Since this is your database, you can determine if a login name and password is required to access the database.
9. If so, then 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, then skip this step.

10. When the ODBC Microsoft Access Setup dialog box appears, select OK.

11. Select OK to close the ODBC Data Source Administrator dialog box.

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, which is a serious drawback in the J2EE environment.

For example, a client that needs to frequently interact 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 available a limited number of connections. 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, which hides from the client details on how the connection pool is maintained.

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 Pooled Connection object that already has a physical connection to the database. Below program 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 J2EE component 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 J2EE component to access the database.

The `close()` method of the `DataSource` object is called once when the J2EE component 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 J2EE component that needs access to the database.

```
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 J2EE component creates and sends a query to access data contained in the database. The query is written using SQL.. 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.



1. The Statement Object

The Statement object is used whenever a J2EE component needs to immediately execute a query 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 query. The ResultSet object also contains methods that are used to manipulate data in the ResultSet. The execute() method of the Statement object is used when there may be multiple results 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 changes values in a row and removes a row respectively.

The executeUpdate() method returns an integer indicating the number of rows that were updated by the query. The executeUpdate() is used to INSERT, UPDATE, DELETE, and DDL statements. The programming enhancements are



to create a query, execute the query, and return a ResultSet. Two new objects are declared in 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 the DBMS to return all the rows from the Customer table of the CustomerInformation database.

```
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();
```

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 J2EE components until the garbage routine is automatically run.

```
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();
```



Java statements used to manipulate the ResultSet is 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 a SQL error message that is passed along to the J2EE component by the executeQuery() method.

The above program illustrates how to use the executeUpdate() method of the Statement object. You'll notice that Listing 6-10 is nearly identical to Listing 6-9. However, the query updates a value in the database rather than requesting that data be returned to the J2EE component. Three changes are made to 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, which can be used for many purposes within the J2EE component such as sending a confirmation notice to the J2EE application that requested database access.

2. PreparedStatement Object

A 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 J2EE component during the same session.

A SQL query can be precompiled and executed by using the PreparedStatement object. In this case, the query is constructed similar to queries. 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. Below program illustrates how to use the PreparedStatement object. 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 that query has a question mark as the value of the customer number. The question mark 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, which is then precompiled.

The setxxx() method of the PreparedStatement object is used to replace the question mark with the value passed to the setxxx() method. There are a number of setxxx() methods available in the PreparedStatement object, each of which specifies the data type of the value that is being passed to the setxxx() method. From the program, the setString() method is used because the customer number is being passed as a string.

The setxxx() 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. 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 setxxx() 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

```
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 briage." + 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.prepareStatement(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();
```

executeUpdate() method.

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

```
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 briage." + 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.prepareStatement(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();
```

3. CallableStatement

The CallableStatement object is used to call a stored procedure from within a J2EE object. A stored procedure's a block of code and is identified by a unique name. The type and style of code depends on the DMBS vendor and can be written in PL/SQL, Transact-SQL, 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 setxxx() 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() then is later retrieved by the J2EE component using the getxxx() method.

The INOUT parameter is a single parameter that is used to both pass information to the stored procedure and retrieve information from a stored procedure using the techniques described in the previous two paragraphs. The below program illustrates how to call a stored procedure and retrieve a value returned by the 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();
```

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.

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 that was requested by the query.

The ResultSet object contains methods that are used to copy data from the ResultSet into a Java collection object or variable 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 J2EE component must move the virtual cursor to each row and 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 getxxx() method is used to copy data from the row to a collection, object, or variable. the getxxx() 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 getxxx() method must be the same data type as the column in the ResultSet. The getxxx() 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

The below code illustrates a commonly used routine to read values from a ResultSet into variables that can later be further processed by the J2EE component. 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, which corresponds 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 if 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

```
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 the 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 J2EE component to specify the number of rows to return from the DBMS.

There are six methods of the ResultSet object that are used to position the virtual cursor in addition to the next() method. 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. The TYPE_SCROLL_INSENSITIVE, and TYPE_SCROLL_SENSITIVE constants permit the virtual cursor to move in both directions. The TYPE_SCROLL_INSENSITIVE constant makes the ResultSet insensitive to changes made by another J2EE component to data in the table whose rows are reflected in the ResultSet. The TYPE_SCROLL_SENSITIVE constant makes the ResultSet sensitive to those changes. The below program illustrates how to reposition the virtual cursor in the



ResultSet. This retrieves customers' first names and last names from the Customers table of the CustomerInformation database. Since the program 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 to 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 8. 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.



```
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 are unable to return a scrollable ResultSet. The below program can be used to test whether or not

```
boolean forward, insensitive, sensitive;
DataBaseMetaData meta = Db.getMetaData();
forward = meta.supportsResultSetType(ResultSet.TYPE_FORWARD_ONLY);
insensitive = meta.supportsResultSetType(ResultSet. TYPE_SCROLL_INSENSITIVE);
sensitive = meta.supportsResultSetType(ResultSet. TYPE_SCROLL_SENSITIVE);
System.out.println("forward: " + answer);
System.out.println("insensitive: " + insensitive);
System.out.println("sensitive: " + sensitive);
(h2) Specify Number Of Rows To Return
```

the JDBC driver in use supports a scrollable ResultSet.

When the J2EE component 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 it. In addition, the maximum row setting is for rows in the ResultSet and not for the number of rows returned by the DBMS.

Updatable ResultSet

Rows contained in the ResultSet can be updatable 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 in which a ResultSet can be changed. 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 ResultSet

Once the executeQuery() method of the Statement object returns a ResultSet, the updateXXX() method is used to change the value of a column in the current row of the ResultSet. The XXX in the updateXXX() method is replaced with the data type of the column that is to be updated. The updateXXX() 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 parameter 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,

```
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);
}
```



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 updatexxx() methods are called. The updateRow() method changes values in columns of the current row of the ResultSet based on the values of the updatexxx() methods. The below program 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 with 'Jones'. The change takes effect once the updateRow() method is called; however, this change only occurs in the ResultSet. The corresponding row in the table remains unchanged until an update query is run.



```
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 = 'Smith'";
    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 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 in the ResultSet that have to be evaluated for the second test. This also deletes it 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 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 Row in the `ResultSet`

Inserting a row into the `ResultSet` is accomplished using basically the same technique as is used to update the `ResultSet`. That is, the `updateXXX()` method is used to specify the column and value that will be placed into the column of the `ResultSet`.

The `updateXXX()` 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 updatexxx() methods, which causes a new row to be inserted into the ResultSet having values that reflect the parameters in the updatexxx() methods. This also updates the underlying database.

The below program 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

```
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";
```

name Tom Smith is inserted into the ResultSet in the third try{} block using the updateString() method.



```
        DataRequest = Db.createStatement(ConCUR_UPDATABLE);
        Results = DataRequest.executeQuery (query);
    }

    catch ( SQLException error ){
        System.out.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"); // updates the ResultSet
        Results.updateString (2, "Smith"); // updates the ResultSet
        Results.insertRow(); // updates the underlying database
        DataRequest.close();
    }
    catch (SQLException error ) {
        System.out.println("Data display error." + error);
        System.exit(5);
    }
}
```

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.

Transaction Processing

A transaction may involve several tasks similar to the tasks that are 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 J2EE component calls the commit() method of the Connection object. All SQL statements executed prior to 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 if the SQL statement is part of a transaction or not. This means that the commit() method must be issued in the previous examples used 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 J2EE component 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

```
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'";
```



again—this timing passing it a true parameter, reactivating the AutoCommit feature.

The below program 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 Customer 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.

```
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());
        }
    }
}
(h2) 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 (i.e., delivered). The confirmation notice can be sent once the email server is back online.

The J2EE component 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.

The below program illustrates how to create a savepoint. A savepoint is created after the execution of the first update SQL statement. There can be many savepoints used in a transaction. Each savepoint 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 more rigorous code is necessary 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.



```
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);
}
(h2) Batch Statements
```

ResultSet Holdability

Whenever the commit() method is called, all ResultSet objects that were created for the transaction are closed. Sometimes a J2EE component 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 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 constant 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.



Metadata

Metadata is data about data, as discussed previously in this chapter. A J2EE component 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 J2EE component 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 username.
- `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

There are two types of metadata that can be retrieved from the DBMS. These are 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 is illustrated in the following code statement:

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

Once the ResultSet metadata is retrieved, the J2EE component can call methods of the ResultSetMetaData object to retrieve specific kinds of



metadata. The more commonly called methods are as follows:

- `getColumnName()` 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`.

Data Types

The `setxxx()` and `getxxx()` methods are used to set a value of a specific data type and to retrieve a value of a specific data type. The `xxx` in the names of these methods is replaced with the name of the data type.



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
FLOAT	float
DOUBLE	double

Exceptions:

There are three kinds of exceptions that are thrown by JDBC methods. These are SQLExceptions, SQLWarnings, and DataTruncation.

SQLExceptions commonly reflect a SQL syntax error in the query and are thrown by many of the methods contained in the java.sql package.

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 DataTrunction exception is thrown.