
Advanced Tutorial

THE `java.sql` package that is included in the Java 2 SDK, Standard Edition, (known as the JDBC 2.0 core API) includes several new features not included in the `java.sql` package that is part of the JDK 1.1 release (referred to as the JDBC 1.0 API). The code samples in the previous chapter use only the JDBC 1.0 API. This chapter shows you how to use the new features in the JDBC 2.0 core API and also how to use the extension API to set up a connection for distributed applications. Although the features described in this chapter are more advanced than those in the “Basic Tutorial” chapter, you do not have to be an advanced programmer to go through this chapter.

In this chapter you will learn to do the following:

- Scroll forward and backward in a result set or move to a specific row
- Make updates to database tables using methods in the Java programming language (instead of using SQL commands)
- Send multiple SQL update statements to the database as a unit, or batch
- Use the new SQL3 data types as column values
- Create new SQL user-defined types (UDTs)
- Map an SQL UDT to a class in the Java programming language
- Make a connection that participates in connection pooling
- Make a connection that can be used for a distributed transaction

3.1 Getting Set Up to Use the JDBC 2.0 API

This section describes what you need to do in order to write or run code that uses features introduced in the JDBC 2.0 API.

3.1.1 Setting Up to Run Code with New Features

To write or run code that employs the new features in the JDBC 2.0 API, you will need to do the following:

1. Download the Java 2 SDK, Standard Edition, following the download instructions
2. Install a JDBC driver that implements the JDBC 2.0 features used in the code
3. Access a DBMS that works with your driver

Your driver vendor may bundle the JDBC Optional Package API (the `javax.sql` package) with its driver product. If it does not, you can download it from the JDBC home page.

<http://java.sun.com/products/jdbc>

NOTE: If you write server side code, you will want to download the Java 2 SDK, Enterprise Edition, instead of the Java 2 SDK, Standard Edition. The Enterprise Edition has the advantage of including the packages `javax.sql`, `javax.naming`, `javax.transaction`, and other extension packages. If you are not writing server side code, however, you will probably want to stick with downloading the leaner Standard Edition.

3.1.2 Using Code Examples

You can download example code from the JDBC web page at

<http://java.sun.com/products/jdbc/book.html>

The code examples for SQL3 functionality are written following the SQL3 standard. All the code has been run on at least one driver, but as of this writing, no one driver implements all of the functionality provided by the JDBC 2.0 API.

Also, some DBMSs use a slightly different syntax for certain operations. For example, the syntax for creating a new data type can vary. At the appropriate point in the tutorial, we show you how to change the generic code we provide for creating a new data type so that it conforms to the syntax your DBMS requires.

Therefore, before you try to run any of the code, check the documentation provided with your driver to see what functionality it supports and what syntax is expected for the operations it performs. Even if it turns out that you cannot run all of the example code with your driver, you still can learn from the examples. Also, the example code you can download includes some additional examples that use syntax specifically for selected drivers. You can download the sample code from the following URL:

<http://java.sun.com/products/jdbc/book.html>

3.2 Moving the Cursor in Scrollable Result Sets

One of the new features in the JDBC 2.0 API is the ability to move a result set's cursor backward as well as forward. There are also methods that move the cursor to a particular row and that check the position of the cursor. Scrollable result sets make it easy to create a graphical interface for browsing result set data, which will probably be one of the main uses for this feature. Another important use is moving the cursor to a row so that you can make updates to that row.

3.2.1 Creating a Scrollable Result Set

Before you can take advantage of these features, however, you need to create a `ResultSet` object that is scrollable. Keep in mind that scrollable result sets involve overhead, so you should create them only when your application uses scrolling. The following code fragment creates a scrollable `ResultSet` object.

```
Statement stmt = con.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery("SELECT COF_NAME, PRICE
                                  FROM COFFEES");
```

This code is similar to what you have used earlier, except that it adds two arguments to the method `createStatement`. The first new argument must be one of the three constants added to the `ResultSet` interface to indicate the type of a `ResultSet` object: `TYPE_FORWARD_ONLY`, `TYPE_SCROLL_INSENSITIVE`, or `TYPE_SCROLL_SENSITIVE`. The second new argument must be one of the two `ResultSet` constants for specifying whether a result set is read-only or updatable: `CONCUR_READ_ONLY` or `CONCUR_UPDATABLE`. The point to remember is that if you specify a result set type, you must also specify whether the result set is read-only or updatable. Also, the order is important. You must specify the type first, and because both parameters are of type `int`, the compiler will not complain if you switch the order.

Specifying the constant `TYPE_FORWARD_ONLY` creates a nonscrollable result set, that is, one in which the cursor moves only forward. If you do not specify any constants for the type and updatability, you will automatically get the default, which is a `ResultSet` object that is `TYPE_FORWARD_ONLY` and `CONCUR_READ_ONLY` (as has always been the case).

To get a scrollable `ResultSet` object, you must specify one of the following `ResultSet` constants: `TYPE_SCROLL_INSENSITIVE` or `TYPE_SCROLL_SENSITIVE`. In some instances, however, specifying one of these constants does not necessarily mean that you will get a scrollable result set. If your driver does not support them, you will get a result set in which the cursor moves forward only. A driver may provide scrollable result sets even if the underlying DBMS does not support them; however, a driver is not required to provide scrolling when the DBMS does not do so. In the end, it is the way your driver is implemented that determines whether you can get a scrollable result set.

The following line of code checks whether the `ResultSet` object `rs` is scrollable.

```
int type = rs.getType();
```

The variable `type` will be one of the following:

1003 to indicate `ResultSet.TYPE_FORWARD_ONLY`

1004 to indicate `ResultSet.TYPE_SCROLL_INSENSITIVE`

1005 to indicate `ResultSet.TYPE_SCROLL_SENSITIVE`

For a larger code example, see “Getting Other Information about a Result Set” on page 190.

The difference between result sets that are `TYPE_SCROLL_INSENSITIVE` and those that are `TYPE_SCROLL_SENSITIVE` has to do with whether they reflect changes that are made to them while they are open and whether certain methods can be called to detect those changes. Generally speaking, a result set that is `TYPE_SCROLL_INSENSITIVE` does not reflect changes made while it is still open, and one that is `TYPE_SCROLL_SENSITIVE` does. All three types of result sets will make changes visible if they are closed and then reopened. At this stage, you do not need to worry about the finer points of a `ResultSet` object’s capabilities, and we will go into a little more detail later.

3.2.2 Moving the Cursor Forward and Backward

Once you have a scrollable `ResultSet` object, `srs` in the example in the previous section, you can use it to move the cursor around in the result set. Remember that when you created a new `ResultSet` object in the previous chapter, it had a cursor positioned before the first row. Even when a result set is scrollable, the cursor is still initially positioned before the first row. In the JDBC 1.0 API, the only way to move the cursor was to call the method `next`. This is still the appropriate method to call when you want to access each row, going from the first row to the last row, but the JDBC 2.0 API adds many other ways to move the cursor.

The counterpart to the method `next`, which moves the cursor forward one row (toward the end of the result set), is the new method `previous`, which moves the cursor backward (one row toward the beginning of the result set). Both methods return `false` when the cursor goes beyond the result set (to the position after the last row or before the first row), which makes it possible to use them in a `while` loop. In the basic tutorial you used the method `next` in a `while` loop, but to refresh your memory, here is an example in which the cursor moves to the first row and then to the next row each time it goes through the `while` loop. The loop ends when the cursor has gone after the last row, causing the method `next` to return `false`. The following code fragment prints out the values in each row of `srs`, with five spaces between the name and price:

```
Statement stmt = con.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery(
    "SELECT COF_NAME, PRICE FROM COFFEES");
```

```

while (srs.next()) {
    String name = srs.getString("COF_NAME");
    float price = srs.getFloat("PRICE");
    System.out.println(name + "    " + price);
}

```

The printout will look something like this:

```

➤ Colombian      7.99
➤ French_Roast   8.99
➤ Espresso       9.99
➤ Colombian_Decaf 8.99
➤ French_Roast_Decaf 9.99

```

As in the following code fragment, you can process all of the rows in *srs* going backward, but to do this, the cursor must start out being after the last row. You can move the cursor explicitly to the position after the last row with the method `afterLast`. From this position, the method `previous` moves the cursor to the last row, and then with each iteration through the `while` loop, it moves the cursor to the previous row. The loop ends when the cursor reaches the position before the first row, where the method `previous` returns `false`.

```

Statement stmt = con.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery(
    "SELECT COF_NAME, PRICE FROM COFFEES");
srs.afterLast();
while (srs.previous()) {
    String name = srs.getString("COF_NAME");
    float price = srs.getFloat("PRICE");
    System.out.println(name + "    " + price);
}

```

The printout will look similar to this:

```

➤ French_Roast_Decaf 9.99
➤ Colombian_Decaf   8.99
➤ Espresso          9.99
➤ French_Roast      8.99
➤ Colombian         7.99

```

As you can see, the printout for each has the same values, but the rows are in the opposite order. For simplicity, we will assume that the DBMS always returns rows in the same order for our sample query.

3.2.3 Moving the Cursor to a Designated Row

You can move the cursor to a particular row in a `ResultSet` object. The methods `first`, `last`, `beforeFirst`, and `afterLast` move the cursor to the position their names indicate. The method `absolute` will move the cursor to the row number indicated in the argument passed to it. If the number is positive, the cursor moves the given number from the beginning, so calling `absolute(1)` puts the cursor on the first row. If the number is negative, the cursor moves the given number from the end, so calling `absolute(-1)` puts the cursor on the last row. The following line of code moves the cursor to the fourth row of `srs`:

```
srs.absolute(4);
```

If `srs` has 500 rows, the following line of code will move the cursor to row 497:

```
srs.absolute(-4);
```

Three methods move the cursor to a position relative to its current position. As you have seen, the method `next` moves the cursor forward one row, and the method `previous` moves the cursor backward one row. With the method `relative`, you can specify how many rows to move from the current row and also the direction in which to move. A positive number moves the cursor forward the given number of rows; a negative number moves the cursor backward the given number of rows. For example, in the following code fragment, the cursor moves to the fourth row, then to the first row, and finally to the third row:

```
srs.absolute(4); // cursor is on the fourth row
. . .
srs.relative(-3); // cursor is on the first row
. . .
srs.relative(2); // cursor is on the third row
```

3.2.4 Getting the Cursor Position

Several methods give you information about the cursor's position.

The method `getRow` lets you check the number of the row where the cursor is currently positioned. For example, you can use `getRow` to verify the position of the cursor in the previous example as follows:

```
srs.absolute(4);
int rowNum = srs.getRow(); // rowNum should be 4
srs.relative(-3);
rowNum = srs.getRow(); // rowNum should be 1
srs.relative(2);
rowNum = srs.getRow(); // rowNum should be 3
```

Four additional methods let you verify whether the cursor is at a particular position. The position is stated in the method names: `isFirst`, `isLast`, `isBeforeFirst`, `isAfterLast`. These methods all return a `boolean` and can therefore be used in a conditional statement.

For example, suppose you have iterated through some rows in a result set and want to print two columns from the current row. To be sure that the cursor has not gone beyond the last row, you could use code such as the following, in which `srs` is a scrollable `ResultSet` object.

```
if (!srs.isAfterLast()) {
    String name = srs.getString("COF_NAME");
    float price = srs.getFloat("PRICE");
    System.out.println(name + "    " + price);
}
```

The preceding code fragment performs as expected because we know that the `ResultSet` object `srs` is not empty. The method `isAfterLast` returns `false` when the cursor is not after the last row and also when the result set is empty, so this code fragment would not have worked correctly if the result set had been empty.

In the next section, you will see how to use the two remaining `ResultSet` methods for moving the cursor, `moveToInsertRow` and `moveToCurrentRow`. You will also see examples illustrating why you might want to move the cursor to certain positions.

3.3 Making Updates to Updatable Result Sets

Another new feature in the JDBC 2.0 API makes JDBC programming easier. This feature is the ability to update rows in a result set using methods in the Java programming language rather than SQL commands.

3.3.1 Creating an Updatable Result Set

Before you can make updates to a `ResultSet` object, you need to create one that is updatable. In order to do this, you supply the `ResultSet` constant `CONCUR_UPDATABLE` to the `createStatement` method. The `Statement` object that is created will produce an updatable `ResultSet` object each time it executes a query. The following code fragment illustrates creating the updatable `ResultSet` object `uprs`. Note that the code also makes `uprs` scrollable. An updatable `ResultSet` object does not necessarily have to be scrollable, but when you are making changes to a result set, you generally want to be able to move around in it. This would be true if, for example, you were editing a form using a graphical user interface (GUI).

```
Connection con = DriverManager.getConnection(
    "jdbc:mySubprotocol:mySubName");
Statement stmt = con.createStatement(
    ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery(
    "SELECT COF_NAME, PRICE FROM COFFEES");
```

The `ResultSet` object `uprs` might look something like this:

COF_NAME	PRICE
-----	-----
Colombian	7.99
French_Roast	8.99
Espresso	9.99
Colombian_Decaf	8.99
French_Roast_Decaf	9.99

We can now use the new JDBC 2.0 methods in the `ResultSet` interface to insert a new row into `uprs`, delete one of its existing rows, or modify one of its column values.

You might note that just specifying that a result set be updatable does not guarantee that the result set you get is updatable. If a driver does not support updatable result sets, it will return one that is readonly. The query you send can also make a difference. In order to get an updatable result set, the query must generally specify the primary key as one of the columns selected, and it should select columns from only one table.

The following line of code checks whether the `ResultSet` object `uprs` is updatable.

```
int concurrency = uprs.getConcurrency();
```

The variable `concurrency` will be one of the following:

```
1007 to indicate ResultSet.CONCUR_READ_ONLY
```

```
1008 to indicate ResultSet.CONCUR_UPDATABLE
```

For a larger code example, see “Getting Other Information about a Result Set” on page 190.

3.3.2 Updating a Result Set Programmatically

An update is the modification of a column value in the current row. Suppose that we want to raise the price of French Roast Decaf coffee to 10.99. Using the JDBC 1.0 API, the update would look something like this:

```
stmt.executeUpdate("UPDATE COFFEES SET PRICE = 10.99 " +
    "WHERE COF_NAME = 'French_Roast_Decaf'");
```

The following code fragment uses the JDBC 2.0 core API to accomplish the same update made in the previous example. In this example `uprs` is the updatable result set generated in the previous section.

```
uprs.last();
uprs.updateFloat("PRICE", 10.99f);
```

Update operations in the JDBC 2.0 API affect column values in the row where the cursor is positioned, so in the first line, the `ResultSet` `uprs` calls the method `last` to move its cursor to the last row (the row where the column `COF_NAME` has

the value 'French_Roast_Decaf'). Once the cursor is on the last row, all of the update methods you call will operate on that row until you move the cursor to another row. The second line changes the value in the PRICE column to 10.99 by calling the method `updateFloat`. This method is used because the column value we want to update is a `float` in the Java programming language.

Note that there is an `f` following the `float` values (as in `10.99f`) to indicate to the Java compiler that the number is a `float`. If the `f` were not there, the compiler would interpret the number as a `double` and issue an error message. This does not apply to the SQL statements sent to the DBMS, which all of our previous updates have been, because they are not compiled by the Java compiler.

The `ResultSet.updateXXX` methods generally take two parameters: the column to update and the new value to put in that column. As with the `ResultSet.getXXX` methods, the parameter designating the column may be either the column name or the column number. There is a different `updateXXX` method for updating each data type (`updateString`, `updateBigDecimal`, `updateInt`, and so on) just as there are different `getXXX` methods for retrieving different data types.

At this point, the price in *uprs* for French Roast Decaf will be 10.99, but the price in the table *COFFEES* in the database will still be 9.99. To make the update take effect in the database, we must call the `ResultSet` method `updateRow`. Here is what the code should look like to update both *uprs* and *COFFEES*:

```
uprs.last();
uprs.updateFloat("PRICE", 10.99f);
uprs.updateRow();
```

Note that you must call the method `updateRow` before moving the cursor. If you move the cursor to another row before calling `updateRow`, the updates are lost, that is, the row will revert to its previous column values.

Suppose that you realize that the update you made is incorrect. You can restore the previous value by calling the `cancelRowUpdates` method if you call it before you have called the method `updateRow`. Once you have called `updateRow`, the method `cancelRowUpdates` will no longer work. The following code fragment makes an update and then cancels it.

```
uprs.last();
uprs.updateFloat("PRICE", 10.99f);
. . .
uprs.cancelRowUpdates();
```


You can do the same thing without using any SQL commands by using new `ResultSet` methods in the JDBC 2.0 API. For example, after you generate a `ResultSet` object containing results from the table `COFFEES`, you can build a new row and then insert it into both the result set and the table `COFFEES` in one step. Every `ResultSet` object has a row called the *insert row*, a special row in which you can build a new row. This row is not part of the result set returned by a query execution; it is more like a separate buffer in which to compose a new row.

The first step is to move the cursor to the insert row, which you do by invoking the method `moveToInsertRow`. The next step is to set a value for each column in the row. You do this by calling the appropriate `updateXXX` method for each value. Note that these are the same `updateXXX` methods you used in the previous section for changing a column value. Finally, you call the method `insertRow` to insert the row you have just populated with values into the result set. This one method simultaneously inserts the row into both the `ResultSet` object and the database table from which the result set was selected.

The following code fragment creates the scrollable and updatable `ResultSet` object `uprs`, which contains all of the rows and columns in the table `COFFEES`.

```
Connection con = DriverManager.getConnection(
    "jdbc:mySubprotocol:mySubName");
Statement stmt = con.createStatement(
    ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery("SELECT * FROM COFFEES");
```

The next code fragment uses the `ResultSet` object `uprs` to insert the row for Kona coffee, shown in the previous SQL code example. It moves the cursor to the insert row, sets the five column values, and inserts the new row into `uprs` and `COFFEES`.

```
uprs.moveToInsertRow();

uprs.updateString("COF_NAME", "Kona");
uprs.updateInt("SUP_ID", 150);
uprs.updateFloat("PRICE", 10.99f);
uprs.updateInt("SALES", 0);
uprs.updateInt("TOTAL", 0);

uprs.insertRow();
```

Because you can use either the column name or the column number to indicate the column to be set, your code for setting the column values could also have looked like this:

```
uprs.updateString(1, "Kona");
uprs.updateInt(2, 150);
uprs.updateFloat(3, 10.99f);
uprs.updateInt(4, 0);
uprs.updateInt(5, 0);
```

You might be wondering why the `updateXXX` methods seem to behave differently here from the way they behaved in the update examples. In those examples, the value set with an `updateXXX` method immediately replaced the column value in the result set. That was true because the cursor was on a row in the result set. When the cursor is on the insert row, the value set with an `updateXXX` method is likewise immediately set, but it is set in the insert row rather than in the result set itself. In both updates and insertions, calling an `updateXXX` method does not affect the underlying database table. The method `updateRow` must be called to have updates occur in the database. For insertions, the method `insertRow` inserts the new row into the result set and the database at the same time.

You might also wonder what happens if you insert a row without supplying a value for every column in the row. If a column has a default value or accepts SQL NULL values, you can get by with not supplying a value. If a column does not have a default value, you will get an `SQLException` if you fail to set a value for it. You will also get an `SQLException` if a required table column is missing in your `ResultSet` object. In the example above, the query was `SELECT * FROM COFFEES`, which produced a result set with all the columns of all the rows. When you want to insert one or more rows, your query does not have to select all rows, but you should generally select all columns. You will normally want to use a `WHERE` clause to limit the number of rows returned by your `SELECT` statement, especially if your table has hundreds or thousands of rows.

After you have called the method `insertRow`, you can start building another row to be inserted, or you can move the cursor back to a result set row. Note that you can move the cursor to another row at any time, but if you move the cursor from the insert row before calling the method `insertRow`, you will lose all of the values you have added to the insert row.

To move the cursor from the insert row back to the result set, you can invoke any of the following methods: `first`, `last`, `beforeFirst`, `afterLast`, `absolute`,

previous, relative, or moveToCurrentRow. When you call the method moveToInsertRow, the result set keeps track of which row the cursor is sitting on, which is, by definition, the current row. As a consequence, the method moveToCurrentRow, which you can invoke only when the cursor is on the insert row, moves the cursor from the insert row back to the row that was previously the current row. This also explains why you can use the methods previous and relative, which require movement relative to the current row.

3.3.4 Code Sample for Inserting a Row

The following code sample is a complete program that you can run if you have a JDBC 2.0 driver that implements scrollable and updatable result sets.

Here are some things you might notice about the code:

1. The `ResultSet` object *uprs* is updatable, scrollable, and sensitive to changes made by itself and others. Even though it is `TYPE_SCROLL_SENSITIVE`, it is possible that the `getXXX` methods called after the insertions will not retrieve values for the newly-inserted rows. There are methods in the `DatabaseMetaData` interface that will tell you what is visible and what is detected in the different types of result sets for your driver and DBMS. (These methods are discussed in detail in Chapter 4, “Metadata Tutorial.”) In this code sample, we wanted to demonstrate cursor movement in the same `ResultSet` object, so after moving to the insert row and inserting two rows, the code moves the cursor back to the result set, going to the position before the first row. This puts the cursor in position to iterate through the entire result set using the method `next` in a while loop. To be absolutely sure that the `getXXX` methods include the inserted row values no matter what driver and DBMS is used, you can close the result set and create another one, reusing the `Statement` object *stmt* with the same query (`SELECT * FROM COFFEES`). A result set opened after a table has been changed will always reflect those changes.
2. After all the values for a row have been set with `updateXXX` methods, the code inserts the row into the result set and the database with the method `insertRow`. Then, still staying on the insert row, it sets the values for another row.

```
import java.sql.*;

public class InsertRows {

    public static void main(String args[]) {

        String url = "jdbc:mySubprotocol:myDataSource";
        Connection con;
        Statement stmt;
        try {
            Class.forName("myDriver.ClassName");

        } catch(java.lang.ClassNotFoundException e) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(e.getMessage());
        }

        try {

            con = DriverManager.getConnection(url,
                "myLogin", "myPassword");

            stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                ResultSet.CONCUR_UPDATABLE);

            ResultSet uprs = stmt.executeQuery("SELECT * FROM COFFEES");

            uprs.moveToInsertRow();

            uprs.updateString("COF_NAME", "Kona");
            uprs.updateInt("SUP_ID", 150);
            uprs.updateFloat("PRICE", 10.99f);
            uprs.updateInt("SALES", 0);
            uprs.updateInt("TOTAL", 0);

            uprs.insertRow();
```

```

uprs.updateString("COF_NAME", "Kona_Decaf");
uprs.updateInt("SUP_ID", 150);
uprs.updateFloat("PRICE", 11.99f);
uprs.updateInt("SALES", 0);
uprs.updateInt("TOTAL", 0);

uprs.insertRow();

uprs.beforeFirst();

System.out.println("Table COFFEES after insertion:");
while (uprs.next()) {
    String name = uprs.getString("COF_NAME");
    int id = uprs.getInt("SUP_ID");
    float price = uprs.getFloat("PRICE");
    int sales = uprs.getInt("SALES");
    int total = uprs.getInt("TOTAL");
    System.out.print(name + "    " + id + "    " + price);
    System.out.println("    " + sales + "    " + total);
}

uprs.close();
stmt.close();
con.close();

} catch(SQLException ex) {
    System.err.println("SQLException: " + ex.getMessage());
}
}
}

```

3.3.5 Deleting a Row Programmatically

So far, you have seen how to update a column value and how to insert a new row. Deleting a row is the third way to modify a `ResultSet` object, and it is the simplest. You simply move the cursor to the row you want to delete and then call the

method `deleteRow`. For example, if you want to delete the fourth row in the `ResultSet` `uprs`, your code will look like this:

```
uprs.absolute(4);
uprs.deleteRow();
```

These two lines of code remove the fourth row from `uprs` and also from the database.

The only issue about deletions is what the `ResultSet` object actually does when it deletes a row. With some JDBC drivers, a deleted row is removed and is no longer visible in a result set. Some JDBC drivers use a blank row as a placeholder (a “hole”) where the deleted row used to be. If there is a blank row in place of the deleted row, you can use the method `absolute` with the original row positions to move the cursor because the row numbers in the result set are not changed by the deletion.

In any case, you should remember that JDBC drivers handle deletions differently. You can use methods in the `DatabaseMetaData` interface to discover the exact behavior of your driver.

3.3.6 Seeing Changes in Result Sets

Result sets vary greatly in their ability to reflect changes made in their underlying data. If you modify data in a `ResultSet` object, the change will always be visible if you close it and then reopen it during a transaction. In other words, if you re-execute the same query after changes have been made, you will produce a new result set based on the new data in the target table. This new result set will naturally reflect changes you made earlier. You will also see changes made by others when you reopen a result set if your transaction isolation level makes them visible.

So when can you see visible changes you or others made while the `ResultSet` object is still open? (Generally, you will be most interested in the changes made by others because you know what changes you made yourself.) The answer depends on the type of `ResultSet` object you have.

With a `ResultSet` object that is `TYPE_SCROLL_SENSITIVE`, you can always see visible updates made to existing column values. You may see inserted and deleted rows, but the only way to be sure is to use `DatabaseMetaData` methods that return this information. (“New JDBC 2.0 Core API Features” on page 371 explains how to ascertain the visibility of changes.)

You can, to some extent, regulate what changes are visible by raising or lowering the transaction isolation level for your connection with the database. For example, the following line of code, where *con* is an active `Connection` object, sets the connection's isolation level to `TRANSACTION_READ_COMMITTED`:

```
con.setTransactionIsolation(
    Connection.TRANSACTION_READ_COMMITTED);
```

With this isolation level, a `TYPE_SCROLL_SENSITIVE` result set will not show any changes before they are committed, but it can show changes that may have other consistency problems. To allow fewer data inconsistencies, you could raise the transaction isolation level to `TRANSACTION_REPEATABLE_READ`. The problem is that, in most cases, the higher the isolation level, the poorer the performance is likely to be. And, as is always true of JDBC drivers, you are limited to the levels your driver actually provides. Many programmers find that the best choice is generally to use their database's default transaction isolation level. You can get the default with the following line of code, where *con* is a newly-created connection:

```
int level = con.getTransactionIsolation();
```

The explanation of `Connection` fields, beginning on page 351, gives the transaction isolation levels and their meanings.

If you want more information about the visibility of changes and transaction isolation levels, see “What Is Visible to Transactions” on page 599.

In a `ResultSet` object that is `TYPE_SCROLL_INSENSITIVE`, you cannot see changes made to it by others while it is still open, but you may be able to see your own changes with some implementations. This is the type of `ResultSet` object to use if you want a consistent view of data and do not want to see changes made by others.

3.3.7 Getting the Most Recent Data

Another new feature in the JDBC 2.0 API is the ability to get the most recent data. You can do this using the method `refreshRow`, which gets the latest values for a row straight from the database. This method can be relatively expensive, especially if the DBMS returns multiple rows each time you call `refreshRow`. Nevertheless, its use can be valuable if it is critical to have the latest data. Even when a result set is sensitive and changes are visible, an application may not always see the very latest changes that have been made to a row if the driver retrieves several rows at a time

and caches them. Thus, using the method `refreshRow` is the only way to be sure that you are seeing the most up-to-date data.

The following code sample illustrates how an application might use the method `refreshRow` when it is absolutely critical to see the most current values. Note that the result set should be sensitive; if you use the method `refreshRow` with a `ResultSet` object that is `TYPE_SCROLL_INSENSITIVE`, `refreshRow` does nothing. (The urgency for getting the latest data is a bit improbable for the table `COFFEES`, but a commodities trader's fortunes could depend on knowing the latest prices in a wildly fluctuating coffee market. Or, for example, you would probably want the airline reservation clerk to check that the seat you are reserving is really still available.)

```
Statement stmt = con.createStatement(
    ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery("SELECT COF_NAME, PRICE
                                   FROM COFFEES");

srs.absolute(4);
float price1 = srs.getFloat("PRICE");
// do something. . .
srs.absolute(4);
srs.refreshRow();
float price2 = srs.getFloat("PRICE");
if (price2 > price1) {
    // do something. . .
}
```

3.4 Making Batch Updates

A batch update is a set of multiple update statements that is submitted to the database for processing as a batch. Sending batch updates can, in some situations, be much more efficient than sending update statements separately. This ability to send updates as a unit, referred to as the batch update facility, is one of the new features provided with the JDBC 2.0 core API.

3.4.1 Using Statement Objects for Batch Updates

In the JDBC 1.0 API, `Statement` objects submit updates to the database individually with the method `executeUpdate`. Multiple `executeUpdate` statements can be sent in the same transaction, but even though they are committed or rolled back as a unit, they are still processed individually. The interfaces derived from `Statement`—`PreparedStatement` and `CallableStatement`—have the same capabilities, using their own versions of `executeUpdate`.

With the JDBC 2.0 core API, `Statement`, `PreparedStatement` and `CallableStatement` objects maintain all of their old functionality and have as an additional feature a list of commands that is associated with them. This list may contain statements for updating, inserting, or deleting a row; and it may also contain DDL statements such as `CREATE TABLE` and `DROP TABLE`. It cannot, however, contain a statement that would produce a `ResultSet` object, such as a `SELECT` statement. In other words, the list can contain only statements that produce an update count.

The list, which is associated with a `Statement` object at its creation, is initially empty. You can add SQL commands to this list with the method `addBatch` and empty it with the method `clearBatch`. When you have finished adding statements to the list, you call the method `executeBatch` to send them all to the database to be executed as a unit, or batch. Now let's see how these methods work.

Let's suppose that our coffee house proprietor wants to start carrying flavored coffees. He has determined that his best source is one of his current suppliers, Superior Coffee, and he wants to add four new coffees to the table `COFFEES`. Because he is inserting only four new rows, a batch update may not improve performance significantly, but this is a good opportunity to demonstrate how to make batch updates. Remember that the table `COFFEES` has five columns: column `COF_NAME` is type `VARCHAR(32)`, column `SUP_ID` is type `INTEGER`, column `PRICE` is type `FLOAT`, column `SALES` is type `INTEGER`, and column `TOTAL` is type `INTEGER`. Each row the proprietor inserts must have values for the five columns in order. The code for inserting the new rows as a batch might look like this:

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

stmt.addBatch("INSERT INTO COFFEES " +
              "VALUES('Amaretto', 49, 9.99, 0, 0)");
stmt.addBatch("INSERT INTO COFFEES " +
              "VALUES('Hazelnut', 49, 9.99, 0, 0)");
stmt.addBatch("INSERT INTO COFFEES " +
```

```

        "VALUES('Amaretto_decaf', 49, 10.99, 0, 0)");
stmt.addBatch("INSERT INTO COFFEES " +
        "VALUES('Hazelnut_decaf', 49, 10.99, 0, 0)");

int [] updateCounts = stmt.executeBatch();
con.commit();
con.setAutoCommit(true);

```

Now let's examine the code line by line.

```
con.setAutoCommit(false);
```

This line disables auto-commit mode for the `Connection` object `con` so that the transaction will not be automatically committed or rolled back when the method `executeBatch` is called. (If you do not recall what a transaction is, you should review the section “Transactions” on page 327.) To allow for correct error handling, you should always disable auto-commit mode before beginning a batch update.

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

This line of code creates the `Statement` object `stmt`. As is true of all newly-created `Statement` objects, `stmt` has an initially empty list of commands associated with it.

```

stmt.addBatch("INSERT INTO COFFEES " +
        "VALUES('Amaretto', 49, 9.99, 0, 0)");
stmt.addBatch("INSERT INTO COFFEES " +
        "VALUES('Hazelnut', 49, 9.99, 0, 0)");
stmt.addBatch("INSERT INTO COFFEES " +
        "VALUES('Amaretto_decaf', 49, 10.99, 0, 0)");
stmt.addBatch("INSERT INTO COFFEES " +
        "VALUES('Hazelnut_decaf', 49, 10.99, 0, 0)");

```

Each of these lines of code adds a command to the list of commands associated with `stmt`. These commands are all `INSERT INTO` statements, each one adding a row consisting of five column values. The values for the columns `COF_NAME` and `PRICE` are self-explanatory. The second value in each row is 49 because that is the

identification number for the supplier, Superior Coffee. The last two values, the entries for the columns SALES and TOTAL, all start out being zero because there have been no sales yet. (SALES is the number of pounds of this row's coffee sold in the current week; TOTAL is the total of all the cumulative sales of this coffee.)

```
int [] updateCounts = stmt.executeBatch();
```

In this line, *stmt* sends the four SQL commands that were added to its list of commands off to the database to be executed as a batch. Note that *stmt* uses the method `executeBatch` to send the batch of insertions, not the method `executeUpdate`, which sends only one command and returns a single update count. The DBMS will execute the commands in the order in which they were added to the list of commands, so it will first add the row of values for Amaretto, then add the row for Hazelnut, then Amaretto decaf, and finally Hazelnut decaf. If all four commands execute successfully, the DBMS will return an update count for each command in the order in which it was executed. The update counts, `int` values indicating how many rows were affected by each command, are stored in the array *updateCounts*.

If all four of the commands in the batch were executed successfully, *updateCounts* will contain four values, all of which are 1 because an insertion affects one row. The list of commands associated with *stmt* will now be empty because the four commands added previously were sent to the database when *stmt* called the method `executeBatch`. You can at any time explicitly empty this list of commands with the method `clearBatch`.

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

The `Connection.commit` method makes the batch of updates to the COFFEES table permanent. This method needs to be called explicitly because the auto-commit mode for this connection was disabled previously.

```
con.setAutoCommit(true);
```

This line of code enables auto-commit mode for the `Connection con`, which is the default. Now each statement will automatically be committed after it is executed, and an application no longer needs to invoke the method `commit`.

The previous code fragment exemplifies a static batch update. It is also possible to have a parameterized batch update, as shown in the following code fragment, where *con* is a Connection object.

```

con.setAutoCommit(false);
PreparedStatement pstmt = con.prepareStatement(
    "INSERT INTO COFFEES VALUES(?, ?, ?, ?, ?)");
pstmt.setString(1, "Amaretto");
pstmt.setInt(2, 49);
pstmt.setFloat(3, 9.99);
pstmt.setInt(4, 0);
pstmt.setInt(5, 0);
pstmt.addBatch();

pstmt.setString(1, "Hazelnut");
pstmt.setInt(2, 49);
pstmt.setFloat(3, 9.99);
pstmt.setInt(4, 0);
pstmt.setInt(5, 0);
pstmt.addBatch();

// ... and so on for each new type of coffee

int [] updateCounts = pstmt.executeBatch();
con.commit();
con.setAutoCommit(true);

```

3.4.2 Batch Update Exceptions

You will get a BatchUpdateException when you call the method `executeBatch` if (1) one of the SQL statements you added to the batch produces a result set (usually a query) or (2) one of the SQL statements in the batch does not execute successfully for some other reason.

You should not add a query (a SELECT statement) to a batch of SQL commands because the method `executeBatch`, which returns an array of update counts, expects an update count from each SQL command that executes successfully. This means that only commands that return an update count (commands such as INSERT INTO, UPDATE, DELETE) or that return 0 (such as CREATE TABLE, DROP TABLE, ALTER TABLE) can be successfully executed as a batch with the `executeBatch` method.

A `BatchUpdateException` contains an array of update counts that is similar to the array returned by the method `executeBatch`. In both cases, the update counts are in the same order as the commands that produced them. This tells you how many commands in the batch executed successfully and which ones they are. For example, if five commands executed successfully, the array will contain five numbers: the first one being the update count for the first command, the second one being the update count for the second command, and so on.

`BatchUpdateException` is derived from `SQLException`. This means that you can use all of the methods available to an `SQLException` object with it. The following code fragment prints all of the `SQLException` information plus the update counts contained in a `BatchUpdateException` object. Because `BatchUpdateException.getUpdateCounts` returns an array of `int`, the code uses a `for` loop to print each of the update counts.

```
try {
    // make some updates
} catch (BatchUpdateException b) {
    System.err.println("----BatchUpdateException----");
    System.err.println("SQLState: " + b.getSQLState());
    System.err.println("Message: " + b.getMessage());
    System.err.println("Vendor: " + b.getErrorCode());
    System.err.print("Update counts: ");
    int [] updateCounts = b.getUpdateCounts();
    for (int i = 0; i < updateCounts.length; i++) {
        System.err.print(updateCounts[i] + " ");
    }
    System.err.println("");
}
```

3.4.3 Sample Code for a Batch Update

The following code puts together the code fragments from previous sections to make a complete program. One thing you might notice is that there are two `catch` blocks at the end of the application. If there is a `BatchUpdateException` object, the first `catch` block will catch it. The second one will catch an `SQLException` object that is not a `BatchUpdateException` object. (All methods will throw an `SQLException` if there is an error accessing data.)

```
import java.sql.*;

public class BatchUpdate {

    public static void main(String args[]) {

        String url = "jdbc:mySubprotocol:myDataSource";
        Connection con;
        Statement stmt;
        try {
            Class.forName("myDriver.ClassName");

        } catch (java.lang.ClassNotFoundException e) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(e.getMessage());
        }

        try {

            con = DriverManager.getConnection(url,
                "myLogin", "myPassword");

            con.setAutoCommit(false);
            stmt = con.createStatement();

            stmt.addBatch("INSERT INTO COFFEES " +
                "VALUES('Amaretto', 49, 9.99, 0, 0)");
            stmt.addBatch("INSERT INTO COFFEES " +
                "VALUES('Hazelnut', 49, 9.99, 0, 0)");
            stmt.addBatch("INSERT INTO COFFEES " +
                "VALUES('Amaretto_decaf', 49, 10.99, 0, 0)");
            stmt.addBatch("INSERT INTO COFFEES " +
                "VALUES('Hazelnut_decaf', 49, 10.99, 0, 0)");

            int [] updateCounts = stmt.executeBatch();
            con.commit();
            con.setAutoCommit(true);
```

```

ResultSet rs = stmt.executeQuery("SELECT * FROM COFFEES");

System.out.println("Table COFFEES after insertion:");
while (rs.next()) {
    String name = rs.getString("COF_NAME");
    int id = rs.getInt("SUP_ID");
    float price = rs.getFloat("PRICE");
    int sales = rs.getInt("SALES");
    int total = rs.getInt("TOTAL");
    System.out.print(name + " " + id + " " + price);
    System.out.println(" " + sales + " " + total);
}

rs.close();
stmt.close();
con.close();

} catch (BatchUpdateException b) {
    System.err.println("----BatchUpdateException----");
    System.err.println("SQLState: " + b.getSQLState());
    System.err.println("Message: " + b.getMessage());
    System.err.println("Vendor: " + b.getErrorCode());
    System.err.print("Update counts: ");
    int [] updateCounts = b.getUpdateCounts();
    for (int i = 0; i < updateCounts.length; i++) {
        System.err.print(updateCounts[i] + " ");
    }
    System.err.println("");

} catch (SQLException ex) {
    System.err.println("----SQLException----");
    System.err.println("SQLState: " + ex.getSQLState());
    System.err.println("Message: " + ex.getMessage());
    System.err.println("Vendor: " + ex.getErrorCode());
}
}
}
}

```

3.5 SQL3 Data Types

The data types commonly referred to as SQL3 (or SQL-99) types are the new data types being adopted in the next version of the ANSI/ISO SQL standard. The JDBC 2.0 API provides interfaces that represent the mapping of these SQL3 data types into the Java programming language. With these new interfaces, you can work with SQL3 data types the same way you do other data types.

The new SQL3 data types give a relational database more flexibility in what can be used as a value for a table column. For example, a column may now be used to store the new type BLOB (Binary Large Object), which can store very large amounts of data as raw bytes. A column may also be of type CLOB (Character Large Object), which is capable of storing very large amounts of data in character format. The new type ARRAY makes it possible to use an array as a column value. Even the new SQL user-defined types (UDTs), structured types and distinct types, can now be stored as column values.

The following list gives the JDBC 2.0 interfaces that map SQL3 types. We will discuss them in more detail later.

- A `Blob` instance maps an SQL BLOB value
- A `Clob` instance maps an SQL CLOB value
- An `Array` instance maps an SQL ARRAY value
- A `Struct` instance maps an SQL structured type value
- A `Ref` instance maps an SQL REF value

3.5.1 DISTINCT Type

There is one more SQL3 data type, the `DISTINCT` type. We consider it separately because it behaves differently from the other SQL3 data types. Being a user-defined type that is based on one of the already existing built-in types, it has no interface as its mapping in the Java programming language. Instead, the standard mapping for a `DISTINCT` type is the Java type to which its underlying SQL type maps.

To illustrate, we will create a `DISTINCT` type and then see how to retrieve, set, or update it. Suppose you always use a two-letter abbreviation for a state and want to create a `DISTINCT` type to be used for these abbreviations. You could define your new `DISTINCT` type with the following SQL statement:

```
CREATE TYPE STATE AS CHAR(2);
```

Some DBMSs use an alternate syntax for creating a DISTINCT type, which is shown in the following line of code:

```
CREATE DISTINCT TYPE STATE AS CHAR(2);
```

If one syntax does not work, you can try the other. Or you can check the documentation for your driver to see the exact syntax it expects.

These statements create a new data type, STATE, which can be used as a column value or as the value for an attribute of an SQL structured type. Because a value of type STATE is in reality a value that is two CHARs, you use the same method to retrieve it that you would use to retrieve a CHAR value, that is, `getString`. For example, assuming that the fourth column of `ResultSet rs` stores values of type STATE, the following line of code retrieves its value.

```
String state = rs.getString(4);
```

Similarly, you would use the method `setString` to store a STATE value in the database and the method `updateString` to modify its value.

3.5.2 Using SQL3 Data Types

You retrieve, store, and update SQL3 data types the same way you do other data types. You use either `ResultSet.getXXX` or `CallableStatement.getXXX` methods to retrieve them, `PreparedStatement.setXXX` methods to store them, and `ResultSet.updateXXX` methods to update them. Probably 90 percent of the operations performed on SQL3 types involve using the `getXXX`, `setXXX`, and `updateXXX` methods. The following table shows which methods to use:

Table 3.1

SQL3 type	getXXX method	setXXX method	updateXXX method
BLOB	<code>getBlob</code>	<code>setBlob</code>	<code>updateBlob</code> *
CLOB	<code>getClob</code>	<code>setClob</code>	<code>updateClob</code> *

Table 3.1

SQL3 type	getXXX method	setXXX method	updateXXX method
ARRAY	getArray	setArray	updateArray *
Structured type	getObject	setObject	updateObject
REF(structured type)	getRef	setRef	updateRef *

* This method will be added to the JDBC API in the next release of the Java 2 Platform. Until that occurs, however, you should use the method `updateObject`, which works just as well.

For example, the following code fragment retrieves an SQL ARRAY value. For this example, suppose that the column `SCORES` in the table `STUDENTS` contains values of type `ARRAY`. The variable `stmt` is a `Statement` object.

```
ResultSet rs = stmt.executeQuery(
    "SELECT SCORES FROM STUDENTS WHERE ID = 002238");
rs.next();
Array scores = rs.getArray("SCORES");
```

The variable `scores` is a logical pointer to the SQL ARRAY object stored in the table `STUDENTS` in the row for student `002238`.

If you want to store a value in the database, you use the appropriate `setXXX` method. For example, the following code fragment, in which `rs` is a `ResultSet` object, stores a `Clob` object:

```
Clob notes = rs.getClob("NOTES");
PreparedStatement pstmt = con.prepareStatement(
    "UPDATE MARKETS SET COMMENTS = ? WHERE SALES < 1000000");
pstmt.setClob(1, notes);
pstmt.executeUpdate();
```

This code sets `notes` as the first parameter in the update statement being sent to the database. The `CLOB` value designated by `notes` will be stored in the table `MARKETS` in column `COMMENTS` in every row where the value in the column `SALES` is less than one million.

3.5.3 Blob, Clob, and Array Objects

An important feature of `Blob`, `Clob`, and `Array` objects is that you can manipulate them without having to bring all of their data from the database server to your client machine. An instance of any of these types is actually a locator (logical pointer) to the object in the database that the instance represents. Because an SQL `BLOB`, `CLOB`, or `ARRAY` object may be very large, this feature can make performance significantly faster.

If you want to bring the data of an SQL `BLOB`, `CLOB`, or `ARRAY` value to the client, you can use methods in the `Blob`, `Clob`, and `Array` interfaces that are provided for this purpose. `Blob` and `Clob` objects materialize the data of the objects they represent as a stream or as a Java array, whereas an `Array` object materializes the SQL `ARRAY` it represents as either a result set or a Java array. For example, after retrieving the SQL `ARRAY` value in the column `ZIPS` as a `java.sql.Array` object, the following code fragment materializes the `ARRAY` value on the client. It then iterates through `zips`, the Java array that contains the elements of the SQL `ARRAY` value, to check that each zip code is valid. This code assumes that the class `ZipCode` has been defined previously with the method `isValid` returning `true` if the given zip code matches one of the zip codes in a master list of valid zip codes.

```
ResultSet rs = stmt.executeQuery("SELECT ZIPS FROM REGIONS");
while (rs.next()) {
    Array z = rs.getArray("ZIPS");
    String[] zips = (String[])z.getArray();
    for (int i = 0; i < zips.length; i++) {
        if (!ZipCode.isValid(zips[i])) {
            . . . // code to display warning
        }
    }
}
```

The preceding example brings out some of the fine points of the `Array` interface. In the following line, the `ResultSet` method `getArray` returns the value stored in the column `ZIPS` of the current row as the `java.sql.Array` object `z`.

```
Array z = rs.getArray("ZIPS");
```

The variable `z` contains a locator, which means that it is a logical pointer to the SQL ARRAY on the server; it does not contain the elements of the ARRAY itself. Being a logical pointer, `z` can be used to manipulate the array on the server.

In the following line, `getArray` is the `Array.getArray` method, not the `ResultSet.getArray` method used in the previous line. Because `Array.getArray` returns an `Object` in the Java programming language and because each zip code is a `String` object, the result is cast to an array of `String` objects before being assigned to the variable `zips`.

```
String[] zips = (String[])z.getArray();
```

The `Array.getArray` method materializes the SQL ARRAY elements on the client as an array of `String` objects. Because, in effect, the variable `zips` contains the elements of the array, it is possible to iterate through `zips` in a `for` loop, looking for zip codes that are not valid.

3.5.4 Creating an SQL Structured Type

SQL structured types and `DISTINCT` types are the two data types that a user can define in SQL. They are often referred to as UDTs (user-defined types), and you create them with an SQL `CREATE TYPE` statement.

Getting back to our example of The Coffee Break, let's suppose that the proprietor has been successful beyond all expectations and has been expanding with new branches. He has decided to add a `STORES` table to his database containing information about each establishment. `STORES` will have four columns: `STORE_NO` for each store's identification number, `LOCATION` for its address, `COF_TYPES` for the coffees it sells, and `MGR` for its manager. The proprietor, now an entrepreneur, opts to make use of the SQL3 data types. Accordingly, he makes the column `LOCATION` be an SQL structured type, the column `COF_TYPES` an SQL ARRAY, and the column `MGR` a `REF(MANAGER)`, with `MANAGER` being an SQL structured type.

The first thing our entrepreneur needs to do is define the new structured types for the address and the manager. An SQL structured type is similar to structured types in the Java programming language in that it has members, called *attributes*, that may be any data type. The entrepreneur writes the following SQL statement to create the new data type `ADDRESS`:

```

CREATE TYPE ADDRESS
(
  NUM INTEGER,
  STREET VARCHAR(40),
  CITY VARCHAR(40),
  STATE CHAR(2),
  ZIP CHAR(5)
);

```

In this definition, the new type ADDRESS has five attributes, which are analogous to fields in a Java class. The attribute NUM is an INTEGER, the attribute STREET is a VARCHAR(40), the attribute CITY is a VARCHAR(40), the attribute STATE is a CHAR(2), and the attribute ZIP is a CHAR(5).

The following code fragment, in which *con* is a valid Connection object, sends the definition of ADDRESS to the DBMS:

```

String createAddress = "CREATE TYPE ADDRESS " +
    "(NUM INTEGER, STREET VARCHAR(40), CITY VARCHAR(40), " +
    "STATE CHAR(2), ZIP CHAR(5))";
Statement stmt = con.createStatement();
stmt.executeUpdate(createAddress);

```

Now ADDRESS is registered with the database as a data type, and our entrepreneur can use it as the data type for a table column or an attribute of a structured type.

3.5.5 Creating a DISTINCT Type

One of the attributes our coffee entrepreneur plans to include in the new structured type MANAGER is the manager's phone number. Because he will always list the phone number as a ten-digit number (to be sure it includes the area code) and will never manipulate it as a number, he decides to define a new type called PHONE_NO that consists of ten characters. The SQL definition of this new DISTINCT type, which can be thought of as a structured type with only one attribute, looks like this:

```

CREATE TYPE PHONE_NO AS CHAR(10);

```

Or, as noted earlier, for some drivers the syntax might look like this:

```

CREATE DISTINCT TYPE PHONE_NO AS CHAR(10);

```

A `DISTINCT` type is always based on another data type, which must be a pre-defined type. In other words, a `DISTINCT` type cannot be based on a UDT. To retrieve or set a value that is a `DISTINCT` type, you use the appropriate method for the underlying type (the type on which it is based). For example, to retrieve an instance of `PHONE_NO`, which is based on a `CHAR`, you would use the method `getString` because that is the method for retrieving a `CHAR`.

Assuming that a value of type `PHONE_NO` is in the fourth column of the current row of the `ResultSet` object `rs`, the following line of code retrieves it.

```
String phoneNumber = rs.getString(4);
```

Similarly, the following line of code sets an input parameter that has type `PHONE_NO` for a prepared statement being sent to the database.

```
pstmt.setString(1, phoneNumber);
```

Adding on to the previous code fragment, the definition of `PHONE_NO` will be sent to the database with the following line of code:

```
stmt.executeUpdate("CREATE TYPE PHONE_NO AS CHAR(10)");
```

After registering the type `PHONE_NO` with the database, our entrepreneur can use it as a column type in a table or, as he wants to do, as the data type for an attribute in a structured type. The definition of `MANAGER` in the following SQL statement uses `PHONE_NO` as the data type for the attribute `PHONE`.

```
CREATE TYPE MANAGER
(
  MGR_ID INTEGER,
  LAST_NAME VARCHAR(40),
  FIRST_NAME VARCHAR(40),
  PHONE PHONE_NO
);
```

Reusing `stmt`, defined previously, the following code fragment sends the definition of the structured type `MANAGER` to the database.

```
String createManager = "CREATE TYPE MANAGER " +
    "(MGR_ID INTEGER, LAST_NAME VARCHAR(40), " +
```

```

        "FIRST_NAME VARCHAR(40), PHONE PHONE_NO");
stmt.executeUpdate(createManager);

```

The following JDBC code, `CreateUDTs.java`, sends the definitions for `ADDRESS`, `MANAGER`, and `PHONE_NO` to the database. If your DBMS uses type names that are different from the data types used in these definitions, you will need to run the program `CreateNewType.java` to create the new types, which is explained immediately following the code for `CreateUDTs.java`. If your driver does not implement `DISTINCT` types, you will need to use `CHAR(10)` as the type for the column `PHONE` and delete the line that creates the type `PHONE_NO` as `CHAR(10)`.

```

import java.sql.*;

public class CreateUDTs {
    public static void main(String args[]) {

        String url = "jdbc:mySubprotocol:myDataSource";
        Connection con;
        Statement stmt;

        String createAddress = "CREATE TYPE ADDRESS (NUM INTEGER, " +
            "STREET VARCHAR(40), CITY VARCHAR(40), " +
            "STATE CHAR(2), ZIP CHAR(5))";

        String createManager = "CREATE TYPE MANAGER (MGR_ID INTEGER, " +
            "LAST_NAME VARCHAR(40), FIRST_NAME VARCHAR(40), " +
            "PHONE PHONE_NO)";

        try {
            Class.forName("myDriver.ClassName");
        } catch (java.lang.ClassNotFoundException e) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(e.getMessage());
        }

        try {

            con = DriverManager.getConnection(url,
                "myLogin", "myPassword");

```

```

        stmt = con.createStatement();

        stmt.executeUpdate(createAddress);
        stmt.executeUpdate("CREATE TYPE PHONE_NO AS CHAR(10)");
        stmt.executeUpdate(createManager);

        stmt.close();
        con.close();

    } catch(SQLException ex) {
        System.err.println("-----SQLException-----");
        System.err.println("SQLState: " + ex.getSQLState());
        System.err.println("Message: " + ex.getMessage());
        System.err.println("Vendor: " + ex.getErrorCode());
    }
}
}
}

```

If your DBMS uses its own DBMS-specific data types, using `CreateUDTs` may not work because it does not use the specific local type names that your DBMS requires. In this case, you can run the code provided in “Sample Code 19” on page 221 to create each UDT individually. This code, `CreateNewType.java`, is very similar to `CreateNewTable.java`, which you probably used to create the tables in the basic tutorial. Instructions for use follow the code.

3.5.6 Using References to Structured Types

Our coffee entrepreneur has created three new data types that he can now use as column types or attribute types in his database: the structured types `LOCATION` and `MANAGER`, and the `DISTINCT` type `PHONE_NO`. He has already used `PHONE_NO` as the type for the attribute `PHONE` in the new type `MANAGER`, and he plans to use `ADDRESS` as the data type for the column `LOCATION` in the table `STORES`. He can use `MANAGER` as the type for the column `MGR`, but he prefers to use the type `REF(MANAGER)` because he often has one person manage two or three stores. By using `REF(MANAGER)` as a column type, he avoids repeating all the data for `MANAGER` when one person manages more than one store.

With the structured type `MANAGER` already created, our entrepreneur can now create a table containing referenceable instances of `MANAGER`. A reference to an

instance of `MANAGER` will have the type `REF(MANAGER)`. An SQL REF is nothing more than a logical pointer to a structured type, so an instance of `REF(MANAGER)` serves as a logical pointer to an instance of `MANAGER`.

Because an SQL REF value needs to be permanently associated with the instance of the structured type that it references, it is stored in a special table together with its associated instance. A programmer does not create REF types directly but rather creates the table that will store referenceable instances of a particular structured type. Every structured type that is to be referenced will have its own table. When you insert an instance of the structured type into the table, the DBMS automatically creates a REF instance. For example, to hold referenceable instances of `MANGER`, our entrepreneur created the following special table using SQL:

```
CREATE TABLE MANAGERS OF MANAGER (OID REF(MANAGER) VALUES ARE
                                   SYSTEM GENERATED);
```

This statement creates a table with the special column `OID`, which stores values of type `REF(MANAGER)`. Each time an instance of `MANAGER` is inserted into the table, the DBMS will generate an instance of `REF(MANAGER)` and store it in the column `OID`. Implicitly, an additional column stores each attribute of `MANAGER` that has been inserted into the table, as well. For example, the following code fragment shows how our entrepreneur created three instances of `MANAGER` to represent three of his managers:

```
INSERT INTO MANAGERS (MGR_ID, LAST_NAME, FIRST_NAME, PHONE) VALUES
(
  000001,
  'MONTROYA',
  'ALFREDO',
  '8317225600'
);

INSERT INTO MANAGERS (MGR_ID, LAST_NAME, FIRST_NAME, PHONE) VALUES
(
  000002,
  'HASKINS',
  'MARGARET',
  '4084355600'
);
```

```

INSERT INTO MANAGERS (MGR_ID, LAST_NAME, FIRST_NAME, PHONE) VALUES
(
    000003,
    'CHEN',
    'HELEN',
    '4153785600'
);

```

The table `MANAGERS` will now have three rows, one row for each manager inserted so far. The column `OID` will contain three unique object identifiers of type `REF(MANAGER)`, one for each instance of `MANAGER`. These object identifiers were generated automatically by the DBMS and will be permanently stored in the table `MANAGERS`. Implicitly an additional column stores each attribute of `MANAGER`. For example, in the table `MANAGERS`, one row contains a `REF(MANAGER)` that references Alfredo Montoya, another row contains a `REF(MANAGER)` that references Margaret Haskins, and a third row contains a `REF(MANAGER)` that references Helen Chen.

To access a `REF(MANAGER)` instance, you select it from its table. For example, our entrepreneur retrieved the reference to Alfredo Montoya, whose ID number is 000001, with the following code fragment:

```

String selectMgr = "SELECT OID FROM MANAGERS WHERE MGR_ID = 000001";
ResultSet rs = stmt.executeQuery(selectMgr);
rs.next();
Ref manager = rs.getRef("OID");

```

Now he can use the variable `manager` as a column value that references Alfredo Montoya.

3.5.7 Sample Code for Creating an SQL REF

The following code example creates the table `MANAGERS`, a table of referenceable instances of the structured type `MANAGER`, and inserts three instances of `MANAGER` into the table. The column `OID` in this table will store instances of `REF(MANAGER)`. After this code is executed, `MANAGERS` will have a row for each of the three `MANAGER` objects inserted, and the value in the `OID` column will be the `REF(MANAGER)` that identifies the instance of `MANAGER` stored in that row.

```
import java.sql.*;

public class CreateRef {

    public static void main(String args[]) {

        String url = "jdbc:mySubprotocol:myDataSource";

        Connection con;
        Statement stmt;
        try {
            Class.forName("myDriver.ClassName");

        } catch(java.lang.ClassNotFoundException e) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(e.getMessage());
        }

        try {
            String createManagers = "CREATE TABLE MANAGERS OF MANAGER " +
                "(OID REF(MANAGER) VALUES ARE SYSTEM GENERATED)";

            String insertManager1 = "INSERT INTO MANAGERS " +
                "(MGR_ID, LAST_NAME, FIRST_NAME, PHONE) VALUES " +
                "(000001, 'MONTTOYA', 'ALFREDO', '8317225600')";

            String insertManager2 = "INSERT INTO MANAGERS " +
                "(MGR_ID, LAST_NAME, FIRST_NAME, PHONE) VALUES " +
                "(000002, 'HASKINS', 'MARGARET', '4084355600')";

            String insertManager3 = "INSERT INTO MANAGERS " +
                "(MGR_ID, LAST_NAME, FIRST_NAME, PHONE) VALUES " +
                "(000003, 'CHEN', 'HELEN', '4153785600')";

            con = DriverManager.getConnection(url,
                "myLogin", "myPassword");

            stmt = con.createStatement();
            stmt.executeUpdate(createManagers);
```


3.5.8 Using SQL3 Types as Column Values

Our entrepreneur now has the UDTs he needs to create the table STORES. He will use the new data types as column types so that he can store instances of the new types in STORES. He will use the structured type ADDRESS as the type for the column LOCATION and the type REF(MANAGER) as the type for the column MGR.

He will also use the SQL3 predefined type ARRAY as a column type. The following line of code creates the type COF_ARRAY as an ARRAY value with ten elements. The base type of COF_ARRAY is VARCHAR(40).

```
CREATE TYPE COF_ARRAY AS ARRAY(10) OF VARCHAR(40);
```

With the new data types defined, the following SQL command creates the table STORES.

```
CREATE TABLE STORES
(
    STORE_NO INTEGER,
    LOCATION ADDRESS,
    COF_TYPES COF_ARRAY,
    MGR REF(MANAGER)
);
```

The type names to use in your JDBC code for creating STORES may be different from those used in the preceding CREATE TABLE statement, depending on the type names used by your DBMS. If you know that your DBMS uses the same type names, you can simply run the program CreateStores.java, shown here. If not, refer to the instructions immediately following CreateStores.java.

```
import java.sql.*;

public class CreateStores {

    public static void main(String args[]) {

        String url = "jdbc:mySubprotocol:myDataSource";
        Connection con;
        String createTable;
        String createArray;
```

```

        createArray = "CREATE TYPE COF_ARRAY AS ARRAY(10) " +
                      "OF VARCHAR(40)";
        createTable = "CREATE TABLE STORES ( " +
                      "STORE_NO INTEGER, LOCATION ADDRESS, " +
                      "COF_TYPER COF_ARRAY, MGR REF(MANAGER))";
        Statement stmt;

        try {
            Class.forName("myDriver.ClassName");

        } catch(java.lang.ClassNotFoundException e) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(e.getMessage());
        }

        try {
            con = DriverManager.getConnection(url,
                                           "myLogin", "myPassword");

            stmt = con.createStatement();

            stmt.executeUpdate(createArray);
            stmt.executeUpdate(createTable);

            stmt.close();
            con.close();

        } catch(SQLException ex) {
            System.err.println("SQLException: " + ex.getMessage());
        }
    }
}

```

If you needed to run `CreateNewTable.java` to create the tables `COFFEES` and `SUPPLIERS`, you should run it to create the table `STORES` as well. As you might recall, this program first prompts you to supply the name and data type for each column in the table. Then it queries the database about the names it uses for each of the data types you supplied and builds a `CREATE TABLE` statement using those

type names. You can see the code and an explanation of it in the section “Generic Applications” on page 209. Note that in `CreateStores`, the type for the column `MGR` is `REF(MANAGER)`, which is the standard format for an SQL `REF` type. Some drivers, however, may require that the code use the format `REF MANAGER` instead.

The following list gives the responses you should type if you run the program `CreateNewTable.java`. Note that the first response is the name of the table. Subsequent responses give the name of a column and then the data type for that column.

```
STORES
STORE_NO
INTEGER
LOCATION
ADDRESS
COF_TYPES
COF_ARRAY
MGR
REF(MANAGER)
```

3.5.9 Inserting SQL3 Types into a Table

The following code fragment inserts one row into the `STORES` table, supplying values for the columns `STORE_NO`, `LOCATION`, `COF_TYPES`, and `MGR`, in that order.

```
INSERT INTO STORES VALUES
(
    100001,
    ADDRESS(888, 'Main_Street', 'Rancho_Alegre', 'CA', '94049'),
    COF_ARRAY('Colombian', 'French_Roast', 'Espresso',
             'Colombian_Decaf', 'French_Roast_Decaf'),
    SELECT OID FROM MANAGERS WHERE MGR_ID = 000001
);
```

Now let’s walk through each column and the value inserted into it.

`STORE_NO`: 100001

This column is type `INTEGER`, and the number 100001 is simply an `INTEGER`, similar to entries we have made before in the tables `COFFEES` and `SUPPLIERS`.

```
LOCATION: ADDRESS(888, 'Main_Street', 'Rancho_Alegre', 'CA',
                '94049')
```

The type for this column is the structured type `ADDRESS`, and this value is the constructor for an instance of `ADDRESS`. When we sent our definition of `ADDRESS` to the DBMS, one of the things it did was to create a constructor for the new type. The comma-separated values in parentheses are the initialization values for the attributes of `ADDRESS`, and they must appear in the same order in which the attributes were listed in the definition of `ADDRESS`. 888 is the value for the attribute `NUM`, which is an `INTEGER`. "Main_Street" is the value for `STREET`, and "Rancho_Alegre" is the value for `CITY`, with both attributes being type `VARCHAR(40)`. The value for the attribute `STATE` is "CA", which is a `CHAR(2)`, and the value for the attribute `ZIP` is "94049", which is a `CHAR(5)`.

```
COF_TYPES: COF_ARRAY('Colombian', 'French_Roast', 'Espresso',
                    'Colombian_Decaf', 'French_Roast_Decaf'),
```

The column `COF_TYPES` is type `COF_ARRAY` with a base type of `VARCHAR(40)`, and the comma-separated values between parentheses are the `String` objects that are the array elements. Our entrepreneur defined the type `COF_ARRAY` as having a maximum of ten elements. This array has five elements because he supplied only five `String` objects for it.

```
MGR: SELECT OID FROM MANAGERS WHERE MGR_ID = 000001
```

The column `MGR` is type `REF(MANAGER)`, which means that a value in this column must be a reference to the structured type `MANAGER`. All of the instances of `MANAGER` are stored in the table `MANAGERS`. All of the instances of `REF(MANAGER)` are also stored in this table, in the column `OID`. The manager for the store described in this row of our table is Alfredo Montoya, and his information is stored in the instance of `MANAGER` that has 100001 for the attribute `MGR_ID`. To get the `REF(MANAGER)` instance associated with the `MANAGER` object for Alfredo Montoya, we select the column `OID` that is in the row where `MGR_ID` is 100001 in the table `MANAGERS`. The value that will be stored in the `MGR` column of `STORES` (the `REF(MANAGER)` value) is the value the DBMS generated to uniquely identify this instance of `MANAGER`.

We can send the preceding SQL statement to the database with the following code fragment:

```
String insertMgr = "INSERT INTO STORES VALUES (100001, " +
    "ADDRESS(888, 'Main_Street', 'Rancho_Alegre', 'CA', '94049'), " +
    "COF_ARRAY('Colombian', 'French_Roast', 'Espresso', " +
        "'Colombian_Decaf', 'French_Roast_Decaf'}), " +
    "SELECT OID FROM MANAGERS WHERE MGR_ID = 000001)";

stmt.executeUpdate(insertMgr);
```

However, because we are going to send several INSERT INTO statements, it will be more efficient to send them all together as a batch update, as in the following code example.

```
import java.sql.*;

public class InsertStores {

    public static void main(String args[]) {

        String url = "jdbc:mySubprotocol:myDataSource";
        Connection con;
        Statement stmt;
        try {

            Class.forName("myDriver.ClassName");

        } catch (java.lang.ClassNotFoundException e) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(e.getMessage());
        }

        try {

            con = DriverManager.getConnection(url,
                "myLogin", "myPassword");

            stmt = con.createStatement();
            con.setAutoCommit(false);
```

```

String insertStore1 = "INSERT INTO STORES VALUES (" +
    "100001, " +
    "ADDRESS(888, 'Main_Street', 'Rancho_Alegre', " +
        "'CA', '94049'), " +
    "COF_ARRAY('Colombian', 'French_Roast', 'Espresso', " +
        "'Colombian_Decaf', 'French_Roast_Decaf'), " +
    "(SELECT OID FROM MANAGERS WHERE MGR_ID = 000001));

stmt.addBatch(insertStore1);

String insertStore2 = "INSERT INTO STORES VALUES (" +
    "100002, " +
    "ADDRESS(1560, 'Alder', 'Ochos_Pinos', " +
        "'CA', '94049'), " +
    "COF_ARRAY('Colombian', 'French_Roast', 'Espresso', " +
        "'Colombian_Decaf', 'French_Roast_Decaf', " +
        "'Kona', 'Kona_Decaf'), " +
    "(SELECT OID FROM MANAGERS WHERE MGR_ID = 000001));

stmt.addBatch(insertStore2);

String insertStore3 = "INSERT INTO STORES VALUES (" +
    "100003, " +
    "ADDRESS(4344, 'First_Street', 'Verona', " +
        "'CA', '94545'), " +
    "COF_ARRAY('Colombian', 'French_Roast', 'Espresso', " +
        "'Colombian_Decaf', 'French_Roast_Decaf', " +
        "'Kona', 'Kona_Decaf'), " +
    "(SELECT OID FROM MANAGERS WHERE MGR_ID = 000002));

stmt.addBatch(insertStore3);

String insertStore4 = "INSERT INTO STORES VALUES (" +
    "100004, " +
    "ADDRESS(321, 'Sandy_Way', 'La_Playa', " +
        "'CA', '94544'), " +
    "COF_ARRAY('Colombian', 'French_Roast', 'Espresso', " +
        "'Colombian_Decaf', 'French_Roast_Decaf', " +
        "'Kona', 'Kona_Decaf'), " +

```

```

"(SELECT OID FROM MANAGERS WHERE MGR_ID = 000002)");

stmt.addBatch(insertStore4);

String insertStore5 = "INSERT INTO STORES VALUES (" +
    "100005, " +
    "ADDRESS(1000, 'Clover_Road', 'Happyville', " +
        "'CA', '90566'), " +
    "COF_ARRAY('Colombian', 'French_Roast', 'Espresso', " +
        "'Colombian_Decaf', 'French_Roast_Decaf'), " +
    "(SELECT OID FROM MANAGERS WHERE MGR_ID = 000003)");

stmt.addBatch(insertStore5);

int [] updateCounts = stmt.executeBatch();

ResultSet rs = stmt.executeQuery("SELECT * FROM STORES");

System.out.println("Table STORES after insertion:");
System.out.println("STORE_NO  LOCATION  COF_TYPE  MGR");
while (rs.next()) {
    int storeNo = rs.getInt("STORE_NO");
    Struct location = (Struct)rs.getObject("LOCATION");
    Object[] locAttrs = location.getAttributes();
    Array coffeeTypes = rs.getArray("COF_TYPE");
    String[] cofTypes = (String[])coffeeTypes.getArray();

    Ref managerRef = rs.getRef("MGR");
    PreparedStatement pstmt = con.prepareStatement(
        "SELECT MANAGER FROM MANAGERS WHERE OID = ?");
    pstmt.setRef(1, managerRef);
    ResultSet rs2 = pstmt.executeQuery();
    rs2.next();
    Struct manager = (Struct)rs2.getObject("MANAGER");
    Object[] manAttrs = manager.getAttributes();

    System.out.print(storeNo + "  ");
    System.out.print(locAttrs[0] + " " + locAttrs[1] + " " +
        locAttrs[2] + ", " + locAttrs[3] + " " +

```

```

        locAttrs[4] + " ");
    for (int i = 0; i < cofTypes.length; i++)
        System.out.print(cofTypes[i] + " ");
    System.out.println(manAttrs[1] + ", " + manAttrs[2]);

    rs2.close();
    pstmt.close();
}

rs.close();
stmt.close();
con.close();

} catch (BatchUpdateException b) {
    System.err.println("-----BatchUpdateException-----");
    System.err.println("SQLState: " + b.getSQLState());
    System.err.println("Message: " + b.getMessage());
    System.err.println("Vendor: " + b.getErrorCode());
    System.err.print("Update counts: ");
    int [] updateCounts = b.getUpdateCounts();
    for (int i = 0; i < updateCounts.length; i++) {
        System.err.print(updateCounts[i] + " ");
    }
    System.err.println("");
} catch (SQLException ex) {
    System.err.println("SQLException: " + ex.getMessage());
    System.err.println("SQLState: " + ex.getSQLState());
    System.err.println("Message: " + ex.getMessage());
    System.err.println("Vendor: " + ex.getErrorCode());
}
}
}

```

3.6 Using Custom Mapping

With business booming, our entrepreneur has found that he is regularly adding new stores and making changes to his database. To make his life a little easier, he has decided to use a custom mapping for the structured type ADDRESS so that he can simply make changes to the Java class that maps ADDRESS. The Java class will have a field for each attribute of ADDRESS, and he can name the class and the fields whatever he wants.

3.6.1 Implementing SQLData

The first thing required for a custom mapping is to create a class that implements the interface SQLData. You will not normally have to do this yourself because tools are being developed to do it. Just so you know what is involved, we will show you what a tool might do. If you want a complete discussion of custom mapping and how it works, see “Creating a Custom Mapping” on page 768.

The SQL definition of the structured type ADDRESS looked like this:

```
CREATE TYPE ADDRESS
(
  NUM INTEGER,
  STREET VARCHAR(40),
  CITY VARCHAR(40),
  STATE CHAR(2),
  ZIP CHAR(5)
);
```

A class generated by a tool to implement SQLData for the custom mapping of ADDRESS might look like this:

```
public class Address implements SQLData {
    public int num;
    public String street;
    public String city;
    public String state;
    public String zip;

    private String sql_type;
    public String getSQLTypeName() {
```

```

        return sql_type;
    }

    public void readSQL(SQLInput stream, String type)
        throws SQLException {
        sql_type = type;
        num = stream.readInt();
        street = stream.readString();
        city = stream.readString();
        state = stream.readString();
        zip = stream.readString();
    }

    public void writeSQL(SQLOutput stream) throws SQLException {
        stream.writeInt(num);
        stream.writeString(street);
        stream.writeString(city);
        stream.writeString(state);
        stream.writeString(zip);
    }
}

```

3.6.2 Using a Connection's Type Map

After writing a class that implements the interface `SQLData`, the only other thing you have to do to set up a custom mapping is to make an entry in a type map. For our example, this means entering the fully-qualified SQL name for `ADDRESS` and the `Class` object for the class `Address`. A type map, an instance of `java.util.Map`, is associated with every new connection when it is created, so we can just use that one. Assuming that `con` is our active connection, the following code fragment adds an entry for the UDT `ADDRESS` to the type map associated with `con`.

```

java.util.Map map = con.getTypeMap();
map.put("SchemaName.ADDRESS", Class.forName("Address"));

```

Whenever we call the `getObject` method to retrieve an instance of `ADDRESS`, the driver will check the type map associated with the connection and see that it has an entry for `ADDRESS`. The driver will note the `Class` object for `Address`, create

an instance of it, and do many other things behind the scenes to map ADDRESS to Address. The nice thing is that you do not have to do anything more than generate the class for the mapping and then make an entry in a type map to let the driver know that there is a custom mapping. The driver will do all the rest.

The situation is similar for storing a structured type that has a custom mapping. When you call the method `setObject`, the driver will check to see if the value to be set is an instance of a class that implements the interface `SQLData`. If it is (meaning that there is a custom mapping), the driver will use the custom mapping to convert the value to its SQL counterpart before returning it to the database. Again, the driver does the custom mapping behind the scenes; all you need to do is supply the method `setObject` with a parameter that has a custom mapping. You will see an example of this later in this section.

Now let's look at the difference between working with the standard mapping, a `Struct` object, and the custom mapping, a class in the Java programming language. The following code fragment shows the standard mapping to a `Struct` object, which is the mapping the driver uses when there is no entry in the connection's type map.

```
ResultSet rs = stmt.executeQuery(
    "SELECT LOCATION WHERE STORE_NO = 100003");
rs.next();
Struct address = (Struct)rs.getObject("LOCATION");
```

The variable `address` contains the following attribute values: 4344, "First_Street", "Verona", "CA", "94545".

The following code fragment shows what happens when there is an entry for the structured type ADDRESS in the connection's type map. Remember that the column LOCATION stores values of type ADDRESS.

```
ResultSet rs = stmt.executeQuery(
    "SELECT LOCATION WHERE STORE_NO = 100003");
rs.next();
Address store_3 = (Address)rs.getObject("LOCATION");
```

The variable `store_3` is now an instance of the class `Address`, with each attribute value being the current value of one of the fields of `Address`. Note that you need to remember to convert the object retrieved by `getObject` to an `Address` object before assigning it to `store_3`. Note also that `store_3` must be an `Address` object.

Now let's compare working with the `Struct` object to working with the instance of `Address`. Suppose the store moved to a better location in the neighboring town and we need to update our database. With the custom mapping, we simply need to reset the fields of `store_3`, as in the following code fragment.

```
ResultSet rs = stmt.executeQuery(
    "SELECT LOCATION WHERE STORE_NO = 100003");
rs.next();
Address store_3 = (Address)rs.getObject("LOCATION");
store_3.num = 1800;
store_3.street = "Artsy_Alley";
store_3.city = "Arden";
store_3.state = "CA";
store_3.zip = "94546";
PreparedStatement pstmt = con.prepareStatement(
    "UPDATE STORES SET LOCATION = ? WHERE STORE_NO = 100003");
pstmt.setObject(1, store_3);
pstmt.executeUpdate();
```

Values in the column `LOCATION` are instances of `ADDRESS`. The driver checks the connection's type map and sees that there is an entry linking `ADDRESS` with the class `Address` and consequently uses the custom mapping indicated in `Address`. When the code calls `setObject` with the variable `store_3` as the second parameter, the driver checks and sees that `store_3` represents an instance of the class `Address`, which implements `SQLData` for the structured type `ADDRESS`, and again automatically uses the custom mapping.

Without a custom mapping for `ADDRESS`, the update would look more like this:

```
PreparedStatement pstmt = con.prepareStatement(
    "UPDATE STORES SET LOCATION.NUM = 1800, " +
    "LOCATION.STREET = 'Artsy_Alley', " +
    "LOCATION.CITY = 'Arden', " +
    "LOCATION.STATE = 'CA', " +
    "LOCATION.ZIP = '94546' " +
    "WHERE STORE_NO = 100003");
pstmt.executeUpdate();
```

3.6.3 Using Your Own Type Map

Up to this point, we have used only the type map associated with a connection for custom mapping. Normally, that is the only type map most programmers will use. However, it is also possible to create a type map and pass it to certain methods so that the driver will use that type map instead of the one associated with the connection. This allows two different mappings for the same UDT. In fact, it is possible to have multiple custom mappings for the same UDT, just as long as each mapping is set up with a class implementing `SQLData` and an entry in a type map. If you do not pass a type map to a method that can accept one, the driver will by default use the type map associated with the connection.

There are very few situations that call for using a type map other than the one associated with a connection. It could be necessary to supply a method with a type map if, for instance, several programmers working on a JDBC application brought their components together and were using the same connection. If two or more programmers had created their own custom mappings for the same SQL UDT, each would need to supply his/her own type map, thus overriding the connection's type map.

3.7 Using Data Sources

This section covers `DataSource` objects, which are the preferred means of getting a connection to a data source. In addition to their other advantages, which will be explained later, `DataSource` objects can provide connection pooling and distributed transactions. This functionality, part of the JDBC 2.0 Optional Package API, is essential for enterprise database computing. In particular, it is integral to Enterprise JavaBeans (EJB) technology.

Using the JDBC 2.0 Optional Package API is not difficult from the programmer's point of view because, in the spirit of EJB, most of the heavy lifting is done for you behind the scenes. This section will show you how to get a connection using the `DataSource` interface and how to make use of distributed transactions and connection pooling. Both of these involve very little change in coding on the part of the application programmer.

The work performed to deploy the classes that make these operations possible, which a system administrator usually does with a tool, varies with the type of `DataSource` object that is being deployed. As a result, most of this section is

devoted to showing how a system administrator sets up the environment so that programmers can use a `DataSource` object to get connections. The other major set of functionality in the JDBC Optional Package API, rowsets, is covered in another chapter. Chapter 5, “Rowset Tutorial,” starting on page 231, explains rowsets and gives examples of what you can do with them.

3.7.1 Using a `DataSource` Object to Get a Connection

In the chapter “Basic Tutorial” you learned how to get a connection using the `DriverManager` class. This section will show you how to use a `DataSource` object to get a connection to your data source, which is the preferred way. You will see why it is better as you go through the rest of this chapter.

A `DataSource` object represents a particular DBMS or some other data source, such as a file. If a company uses more than one data source, it will deploy a separate `DataSource` object for each of them. A `DataSource` object may be implemented in three different ways:

1. A basic `DataSource` implementation—produces standard `Connection` objects that are not pooled or used in a distributed transaction
2. A `DataSource` class that supports connection pooling—produces `Connection` objects that participate in connection pooling, that is, connections that can be recycled
3. A `DataSource` class that supports distributed transactions—produces `Connection` objects that can be used in a distributed transaction, that is, a transaction that accesses two or more DBMS servers

A driver that supports the JDBC 2.0 API should include at least a basic `DataSource` implementation. A `DataSource` class that supports distributed transactions typically also implements support for connection pooling. For example, a `DataSource` class provided by an EJB vendor will almost always support both connection pooling and distributed transactions.

Let’s assume that the owner of the thriving chain of The Coffee Break shops, from our previous examples, has decided to expand further by selling coffee over the internet. With the amount of traffic he expects, he will definitely need connection pooling. Opening and closing connections involves a great deal of overhead, and he anticipates that his online ordering system will necessitate a sizable number of queries and updates. With connection pooling, a pool of connections can be used over and over again, avoiding the expense of creating a new connection for

every database access. In addition, he now has a second DBMS that contains data for the coffee roasting company he has just acquired. This means that he will want to be able to write distributed transactions that use both his old DBMS server and the new one.

Our entrepreneur has reconfigured his computer system to serve his new, larger customer base. He has bought a JDBC driver that supports all of the JDBC 2.0 API. He has also bought an EJB application server that works with the JDBC 2.0 API to be able to use distributed transactions and get the increased performance that comes with connection pooling. Because of the JDBC 2.0 API, he can choose from a variety of JDBC drivers that are compatible with the EJB server he has purchased. He now has a three-tier architecture, with his new EJB application server and JDBC driver in the middle tier and the two DBMS servers as the third tier. Client machines making requests are the first tier.

Now he needs to have his system administrator, SoLan, deploy the `DataSource` objects so that he and his programmers can start using them. Deploying a `DataSource` object consists of three tasks:

1. Creating an instance of the `DataSource` class
2. Setting its properties
3. Registering it with a naming service that uses the Java Naming and Directory Interface (JNDI) API

The next section will walk you through these steps.

3.7.2 Deploying a Basic `DataSource` Object

First, let's consider the most basic case, which is to use a basic implementation of the `DataSource` interface, that is, one that does not support connection pooling or distributed transactions. In this case there is only one `DataSource` object that needs to be deployed. A basic implementation of `DataSource` produces the same kind of connections that the `DriverManager` produces.

Suppose a company that wants only a basic implementation of `DataSource` has bought a driver from the JDBC vendor DB Access, Inc., that includes the class `com.dbaccess.BasicDataSource`. Now let's look at some code that creates an instance of the class `BasicDataSource` and sets its properties. After the instance of `BasicDataSource` is deployed, a programmer can call the method `DataSource.getConnection` on it to get a connection to the company's database,

CUSTOMER_ACCOUNTS. First, the system administrator creates the `BasicDataSource` object `ds` using the default constructor; then she sets three properties. Note that the code shown here is code that will typically be executed by a tool.

```
com.dbaccess.BasicDataSource ds = new com.dbaccess.BasicData-
Source();
ds.setServerName("grinder");
ds.setDatabaseName("CUSTOMER_ACCOUNTS");
ds.setDescription("Customer accounts database for billing");
```

The variable `ds` now represents the database `CUSTOMER_ACCOUNTS` installed on the server `grinder`. Any connection produced by `ds` will be a connection to the database `CUSTOMER_ACCOUNTS`. With the properties set, the system administrator can register the `BasicDataSource` object with a JNDI naming service. The particular naming service that is used is usually determined by a system property, which is not shown here. Let's look at the code that registers the `BasicDataSource` object, binding it with the logical name `jdbc/billingDB`.

```
Context ctx = new InitialContext();
ctx.bind("jdbc/billingDB", ds);
```

This code uses the JNDI API. The first line creates an `InitialContext` object, which serves as the starting point for a name, similar to `root` in a directory file system. The second line associates, or binds, the `BasicDataSource` object `ds` to the logical name `jdbc/billingDB`. Later you will see that you can give the naming service this logical name, and it will return the `BasicDataSource` object. The logical name can be almost anything you want. In this case, the company decided to use the name `billingDB` as the logical name for the `CUSTOMER_ACCOUNTS` database.

In this example, `jdbc` is a subcontext under the initial context, just as a directory under the root directory is a subdirectory. You can think of `jdbc/billingDB` as being like a path name, where the last item in the path is analogous to a file name. In our case, `billingDB` is the logical name we want to give to `ds`. The subcontext `jdbc` is reserved for logical names to be bound to `DataSource` objects, so `jdbc` will always be the first part of a logical name for a data source.

After a basic `DataSource` implementation is deployed by a system administrator, it is ready for a programmer to use. This means that a programmer can give the logical data source name that was bound to an instance of a `DataSource` class, and the JNDI naming service will return an instance of that `DataSource` class. The

method `getConnection` can then be called on that `DataSource` object to get a connection to the data source it represents. For example, a developer might write the following two lines of code to get a `DataSource` object that will produce a connection to the database `CUSTOMER_ACCOUNTS`.

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("jdbc/billingDB");
```

The first line of code gets an initial context as the starting point for retrieving a `DataSource` object. When we supply the logical name `jdbc/billingDB` to the method `lookup`, it returns the `DataSource` object that the system administrator bound to `jdbc/billingDB` at deployment time. Because the return value of the method `lookup` is a Java Object, we need to cast it to the more specific `DataSource` type before assigning it to the variable `ds`.

The variable `ds` is an instance of the class `com.dbaccess.BasicDataSource` that implements the `DataSource` interface. Calling the method `getConnection` on `ds` will produce a connection to `CUSTOMER_ACCOUNTS`.

```
Connection con = ds.getConnection("fernanda", "brewed");
```

Only the user name and password need to be passed to the `getConnection` method because `ds` has the rest of the information necessary for establishing a connection with `CUSTOMER_ACCOUNTS`, such as the database name and location, in its properties.

Because of its properties, a `DataSource` object is a better alternative than the `DriverManager` facility for getting a connection. For one thing, programmers no longer have to hard code the driver name or JDBC URL in their applications, which makes them more portable. Also, `DataSource` properties make maintaining code much simpler. If there is a change, the system administrator can simply update the data source's properties, and you don't have to worry about changing every application that makes a connection to the data source. For example, if the data source was moved to a different server, all the system administrator would need to do is set the `serverName` property to the new server name.

Aside from portability and ease of maintenance, using a `DataSource` object to get connections can offer other advantages. When a `DataSource` class is implemented to work with a `ConnectionPoolDataSource` implementation, all of the connections produced by instances of that `DataSource` class will automatically be pooled connections. Similarly, when a `DataSource` class is implemented to work with an `XADataSource` class, all of the connections it produces will automatically

be connections that can be used in a distributed transaction. The next section shows how to deploy these types of `DataSource` implementations.

3.7.3 Deploying Other `DataSource` Implementations

A system administrator or another person working in that capacity can deploy a `DataSource` object so that the connections it produces are pooled connections. To do this, he/she first deploys a `ConnectionPoolDataSource` object and then deploys a `DataSource` object implemented to work with it. The properties of the `ConnectionPoolDataSource` object are set so that it represents the data source to which connections will be produced. After the `ConnectionPoolDataSource` object has been registered with a JNDI naming service, the `DataSource` object is deployed. Generally only two properties need to be set for the `DataSource` object: `description` and `dataSourceName`. The value given to the `dataSourceName` property is the logical name identifying the `ConnectionPoolDataSource` object previously deployed, which is the object containing the properties needed to make the connection.

If you are interested in what happens internally, you can refer to Chapter 14, “`ConnectionPoolDataSource`.” Basically, with the `ConnectionPoolDataSource` and `DataSource` objects deployed, you can call the method `DataSource.getConnection` on the `DataSource` object and get a pooled connection. This connection will be to the data source specified in the `ConnectionPoolDataSource` object’s properties.

Let’s look at how a system administrator would deploy a `DataSource` object implemented to provide pooled connections. The system administrator would typically use a deployment tool, so the code fragments shown in this section are the code that a deployment tool would execute.

Let’s go back to our entrepreneur to make things more concrete. To get better performance, The Coffee Break company has bought a driver from DB Access, Inc., that includes the class `com.dbaccess.ConnectionPoolDS`, which implements the `ConnectionPoolDataSource` interface. SoLan, the system administrator, will create an instance of this class, set its properties, and register it with a JNDI naming service. The Coffee Break has bought its `DataSource` class, `com.applogic.PooledDataSource`, from its EJB server vendor, Application Logic, Inc. The class `com.applogic.PooledDataSource` implements connection pooling by using the underlying support provided by classes that implement the interface `ConnectionPoolDataSource`, such as `com.dbaccess.ConnectionPoolDS`.

Interface	Class
ConnectionPoolDataSource	com.dbaccess.ConnectionPoolDS
DataSource	com.applogic.PooledDataSource

The `ConnectionPoolDataSource` object needs to be deployed first. Here is the code to create an instance of `com.dbaccess.ConnectionPoolDS` and set its properties.

```
com.dbaccess.ConnectionPoolDS cpds =
    new com.dbaccess.ConnectionPoolDS();
cpds.setServerName("creamer");
cpds.setDatabaseName("COFFEEBREAK");
cpds.setPortNumber(9040);
cpds.setDescription("Connection pooling for COFFEEBREAK DBMS");
```

After the `ConnectionPoolDataSource` object has been deployed, SoLan will deploy the `DataSource` object. Here is the code for registering the `com.dbaccess.ConnectionPoolDS` object `cpds` with a JNDI naming service. Note that the logical name being associated with `cpds` has the subcontext `pool` added under the subcontext `jdbc`, which is similar to adding a subdirectory to another subdirectory in a hierarchical file system. The logical name of any instance of the class `com.dbaccess.ConnectionPoolDS` will always begin with `jdbc/pool`. We recommend putting all `ConnectionPoolDataSource` objects under the subcontext `jdbc/pool`.

```
Context ctx = new InitialContext();
ctx.bind("jdbc/pool/fastCoffeeDB", cpds);
```

Now it is time to deploy the `DataSource` class that is implemented to interact with `cpds`, an object that implements `ConnectionPoolDataSource`. Here is the code for creating an instance and setting its properties. Note that only two properties are set for this instance of `com.applogic.PooledDataSource`. The `description` property is set because it is always required. The other property that is set, `dataSourceName`, gives the logical JNDI name for `cpds`. In this case, the logical name is `"jdbc/pool/fastCoffeeDB"`, which was bound to `cpds` in the preceding code fragment. Being an instance of the class `com.dbaccess.ConnectionPoolDS`,

cpds represents the `ConnectionPoolDataSource` object that will implement connection pooling for our `DataSource` object.

The following code fragment, which would probably be executed by a deployment tool, creates a `PooledDataSource` object, sets its properties, and binds it to the logical name `jdbc/fastCoffeeDB`.

```
com.appllogic.PooledDataSource ds =
    new com.appllogic.PooledDataSource();
ds.setDescription("produces pooled connections to COFFEEBREAK");
ds.setDataSourceName("jdbc/pool/fastCoffeeDB");

Context ctx = new InitialContext();
ctx.bind("jdbc/fastCoffeeDB", ds);
```

We now have a `DataSource` object deployed that an application can use to get pooled connections to the database `COFFEEBREAK`.

3.7.4 Getting and Using a Pooled Connection

Now that these `DataSource` and `ConnectionPoolDataSource` objects are deployed, an application programmer can use the `DataSource` object to get a pooled connection. The code for getting a pooled connection is just like the code for getting a non-pooled connection, as shown in the following two lines.

```
ctx = new InitialContext();
ds = (DataSource)ctx.lookup("jdbc/fastCoffeeDB");
```

The variable *ds* represents a `DataSource` object that will produce a pooled connection to the database `COFFEEBREAK`. We need to retrieve this `DataSource` object only once because we can use it to produce as many pooled connections as we need. Calling the method `getConnection` on *ds* will automatically produce a pooled connection because the `DataSource` object that *ds* represents was configured to produce pooled connections.

As explained in “Application Code for Connection Pooling” on page 529, connection pooling is generally transparent to the application programmer. There are only two things you need to do when you are using pooled connections:

1. Use a `DataSource` object rather than the `DriverManager` class to get a connection. In the following line of code, `ds` is a `DataSource` object implemented and deployed so that it will create pooled connections.

```
Connection con = ds.getConnection("myLogin", "myPassword");
```

2. Use a `finally` statement to close a pooled connection. The following `finally` statement would appear after the `try/catch` block that applies to the code in which the pooled connection was used.

```
try {
    Connection con = ds.getConnection("myLogin", "myPassword");

    // ... code to use the pooled connection con

} catch (Exception ex {
    // . . . code to handle exceptions
} finally {

    if(con != null) con.close();
}
```

Otherwise, an application using a pooled connection is identical to an application using a regular connection. The only other thing an application programmer might notice when connection pooling is being done is that performance is better.

The following sample code gets a `DataSource` object that produces connections to the database `COFFEEBREAK` and uses it to update a price in the table `COFFEES`. To summarize, the connection in the following code sample participates in connection pooling because the following are true:

- An instance of a class implementing `ConnectionPoolDataSource` has been deployed
- An instance of a class implementing `DataSource` has been deployed, and the value set for its `dataSourceName` property is the logical name that was bound to the previously-deployed `ConnectionPoolDataSource` object

Note that although this code is very similar to code you have seen before, it is different in the following ways:

- It imports the `javax.sql`, `javax.ejb`, and `javax.naming` packages in addition to `java.sql`

The `DataSource` and `ConnectionPoolDataSource` interfaces are in the `javax.sql` package, and the JNDI constructor `InitialContext` and method `Context.lookup` are part of the `javax.naming` package. This particular example code is in the form of an EJB component (an enterprise Bean) that uses API from the `javax.ejb` package. The purpose of this example is to show that you use a pooled connection the same way you use a non-pooled connection, so you need not worry about understanding the EJB API. If you are curious about it, you can go to “An EJB Example” on page 240 for an example and more information.

- It uses a `DataSource` object to get a connection instead of using the `Driver-Manager` facility
- It uses a `finally` statement to be sure that the connection is closed

The following code example obtains and uses a pooled connection, which is used in exactly the same way a regular connection is used.

```
import java.sql.*;
import javax.sql.*;
import javax.ejb.*;
import javax.naming.*;

public class ConnectionPoolingBean implements SessionBean {

    // ...

    public void ejbCreate () throws CreateException {
        ctx = new InitialContext();
        ds = (DataSource)ctx.lookup("jdbc/fastCoffeeDB");
    }

    public void updatePrice(float price, String cofName)
        throws SQLException{

        Connection con;
        PreparedStatement pstmt;
```

```

try {
    con = ds.getConnection("webLogin", "webPassword");
    con.setAutoCommit(false);
    pstmt = con.prepareStatement("UPDATE COFFEES " +
        "SET PRICE = ? WHERE COF_NAME = ?");
    pstmt.setFloat(1, price);
    pstmt.setString(2, cofName);
    pstmt.executeUpdate();

    con.commit();

    pstmt.close();

} finally {
    if (con != null) con.close();
}

private DataSource ds = null;
private Context ctx = null;
}

```

So far you have seen that an application programmer can get a pooled connection without doing anything different. When someone acting as a system administrator has deployed a `ConnectionPoolDataSource` object and a `DataSource` object properly, an application simply uses that `DataSource` object to get a pooled connection. You have also seen that using a pooled connection is just like using a regular connection. An application should, however, use a `finally` clause to close the pooled connection. For simplicity in the preceding code example, we used a `finally` block but no `catch` block. If an exception is thrown by a method in the `try` block, it will be thrown by default, and the `finally` clause will be executed in any case.

3.7.5 Deployment for Distributed Transactions

This section shows how to deploy `DataSource` objects for getting connections that can be used in distributed transactions. As with connection pooling, two different

class instances must be deployed: an `XADataSource` object and a `DataSource` object that is implemented to work with it.

Suppose that the EJB server that our entrepreneur bought includes the `DataSource` class `com.applogic.Transactiona1DS`, which works with an `XADataSource` class such as `com.dbaccess.XATransactiona1DS`. The fact that it works with any `XADataSource` class makes the EJB server portable across JDBC drivers. When the `DataSource` and `XADataSource` objects are deployed, the connections produced will be able to participate in distributed transactions. In this case, the class `com.applogic.Transactiona1DS` is implemented so that the connections produced are also pooled connections, which will usually be the case for `DataSource` classes provided as part of an EJB server implementation.

The `XADataSource` object needs to be deployed first. Here is the code to create an instance of `com.dbaccess.XATransactiona1DS` and set its properties.

```
com.dbaccess.XATransactiona1DS xads =
    new com.dbaccess.XATransactiona1DS();
xads.setServerName("creamer");
xads.setDatabaseName("COFFEEBREAK");
xads.setPortNumber(9040);
xads.setDescription(
    "Distributed transactions for COFFEEBREAK DBMS");
```

Here is the code for registering the `com.dbaccess.XATransactiona1DS` object `xads` with a JNDI naming service. Note that the logical name being associated with `xads` has the subcontext `xa` added under `jdbc`. We recommend that the logical name of any instance of the class `com.dbaccess.XATransactiona1DS` always begin with `jdbc/xa`.

```
Context ctx = new InitialContext();
ctx.bind("jdbc/xa/distCoffeeDB", xads);
```

Now it is time to deploy the `DataSource` object that is implemented to interact with `xads` and other `XADataSource` objects. Note that our `DataSource` class, `com.applogic.Transactiona1DS`, can work with an `XADataSource` class from any JDBC driver vendor. Deploying the `DataSource` object involves creating an instance of `com.applogic.Transactiona1DS` and setting its properties. The `dataSourceName` property is set to `jdbc/xa/distCoffeeDB`, the logical name associated with `com.dbaccess.XATransactiona1DS`. This is the `XADataSource` class that

implements the distributed transaction capability for our `DataSource` class. The following code fragment deploys an instance of our `DataSource` class.

```
com.applogic.Transactiona1DS ds =
                                new com.applogic.Transactiona1DS();
ds.setDescription(
    "Produces distributed transaction connections to COFFEEBREAK");
ds.setDataSourceName("jdbc/xa/distCoffeeDB");

Context ctx = new InitialContext();
ctx.bind("jdbc/distCoffeeDB", ds);
```

Now that we have deployed instances of the classes `com.applogic.Transactiona1DS` and `com.dbaccess.XATransactiona1DS`, an application can call the method `getConnection` on instances of `Transactiona1DS` to get a connection to the COFFEEBREAK database that can be used in distributed transactions.

3.7.6 Using Connections for Distributed Transactions

To get a connection that can be used for distributed transactions, you need to use a `DataSource` object that has been properly implemented and deployed, as shown in the preceding section. With such a `DataSource` object, you simply call the method `getConnection` on it. Once you have the connection, you use it just as you would use any other connection. Because "jdbc/distCoffeeDB" has been associated with an `XADataSource` object in a JNDI naming service, the following code fragment produces a `Connection` object that can be used in distributed transactions.

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("jdbc/distCoffeeDB");
Connection con = ds.getConnection();
```

There are some minor but important restrictions on how this connection is used while it is part of a distributed transaction, however. A transaction manager controls when a distributed transaction begins and when it is committed or rolled back; therefore, application code should never call the methods `Connection.commit` or `Connection.rollback`. An application should likewise never call `Connection.setAutoCommit(true)`, which enables auto-commit mode, because that would also interfere with the transaction manager's control of the transaction boundaries. This explains why a new connection that is created in the scope of a

distributed transaction has its auto-commit mode disabled by default. Note that these restrictions apply only when a connection is participating in a distributed transaction; there are no restrictions while the connection is not part of a distributed transaction.

For the following example, suppose that an order of coffee has been shipped, which triggers updates to two tables that reside on different DBMS servers. The first table is a new INVENTORY table, and the second is the COFFEES table. Because these tables are on different DBMS servers, a transaction that involves both of them will be a distributed transaction. The code in the following example, which obtains a connection, updates the COFFEES table, and closes the connection, is the second part of a distributed transaction.

Note that the code does not explicitly commit or rollback the updates because the scope of the distributed transaction is being controlled by the middle tier server's underlying system infrastructure. Also, assuming that the connection used for the distributed transaction is a pooled connection, the application uses a finally clause to close the connection. This guarantees that a valid connection will be closed even if an exception is thrown, thereby ensuring that the connection is recycled.

The following code sample illustrates an enterprise Bean, a class that implements the methods that can be called by a client. The purpose of this example is to demonstrate that application code for a distributed transaction is no different from other code except that it does not call the Connection methods commit, rollback, or setAutoCommit(true). Therefore, you do not need to worry about understanding the EJB API that is used. If you are interested, the section "Distributed Transactions and EJB" on page 894 describes the scope of transactions in EJB applications. "An EJB Example" on page 240 explains the parts of an EJB application.

```
import java.sql.*;
import javax.sql.*;
import javax.ejb.*;
import javax.naming.*;

public class DistributedTransactionBean implements SessionBean {

    // ...

    public void ejbCreate () throws CreateException {
```

```

        ctx = new InitialContext();
        ds = (DataSource)ctx.lookup("jdbc/distCoffeesDB");
    }

    public void updateTotal(int incr, String cofName)
        throws SQLException {

        Connection con;
        PreparedStatement pstmt;
        try {
            con = ds.getConnection("webLogin", "webPassword");
            pstmt = con.prepareStatement("UPDATE COFFEES " +
                "SET TOTAL = TOTAL + ? WHERE COF_NAME = ?");
            pstmt.setInt(1, incr);
            pstmt.setString(2, cofName);
            pstmt.executeUpdate();
            stmt.close();

        } finally {
            if (con != null) con.close();
        }
    }
    private DataSource ds = null;
    private Context ctx = null;
}

```

Congratulations! You have completed your walk through the new functionality added in the JDBC 2.0 core API (the `java.sql` package), and you have also learned how to use the `javax.sql` package API for getting a connection from a `DataSource` object. You can now create scrollable and updatable result sets, move the cursor many different ways, make updates programmatically, send batch updates, create UDTs, use SQL3 types, and do custom mapping of UDTs. In addition, you know how to write code that uses connection pooling and distributed transactions. Chapter 5, “RowSet Tutorial,” shows you how to use a rowset to provide updating and scrollability, and it also gives an example of using a rowset in an EJB component.