

JDBC

Lesson: JDBC Basics

In this lesson you will learn the basics of the JDBC API. We start by giving you set up instructions in [Getting Started](#) , [Setting Up a Database](#) , and [Establishing a Connection](#) . The next sections discuss how to create and update tables, use joins, transactions and stored procedures. The final sections give instructions on how to complete your JDBC application and how to convert it to an applet.


This lesson covers the JDBC 1.0 API, which is included in JDK tm 1.1, and note where procedures have changed in JDBC 2.0, which is included in JDK 1.2. For coverage of JDBC 2.0 and more advanced features, see the next lesson, [New Features in the JDBC 2.0 API](#).

Note: Most JDBC drivers available at the time of this printing are for JDBC 1.0. More drivers will become available for JDBC 2.0 as it gains wider acceptance.

Getting Set Up to Use the JDBC 2.0 API

If you want to run code that employs any of the JDBC 2.0 features, you will need to do the following:

1. Download JDK 1.2, 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 implements the JDBC 2.0 features used in the code

If your driver does not bundle the JDBC 2.0 Standard Extension API (the `javax.sql` package, you can download it from the [JDBC home page](#).

The JDBC 2.0 API enables you 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 statements to the database as a unit, or a batch.

Getting Started

Setting Up a Database

Establishing a Connection

Setting Up Tables

Retrieving Values from Result Sets

Updating Tables

Milestone: The Basics of JDBC

Using Prepared Statements

Using Joins

Using Transactions

Stored Procedures

SQL Statements for Creating a Stored Procedure

Creating Complete JDBC Applications

Running the Sample Applications

Creating an Applet from an Application

Getting Started

The first thing you need to do is check that you are set up properly. This involves the following steps:

1. Install Java and JDBC on your machine.

To install both the Java tm platform and the JDBC API, simply follow the instructions for downloading the latest release of the JDK tm (Java Development Kit tm). When you download the JDK, you will get JDBC as well. The sample code demonstrating the JDBC 1.0 API was written for JDK1.1 and will run on any version of the Java platform that is compatible with JDK1.1, including JDK1.2. Note that the sample code illustrating the JDBC 2.0 API requires JDK1.2 and will not run on JDK1.1.

You can find the latest release (JDK1.2 at the time of this writing) at the following URL:

<http://java.sun.com/products/JDK/CurrentRelease>

2. Install a driver on your machine.

Your driver should include instructions for installing it. For JDBC drivers written for specific DBMSs, installation consists of just copying the driver onto your machine; there is no special configuration needed.

The JDBC-ODBC Bridge driver is not quite as easy to set up. If you download either the Solaris or Windows versions of JDK1.1, you will automatically get the JDBC-ODBC Bridge driver, which does not itself require any special configuration. ODBC, however, does. If you do not already have ODBC on your machine, you will need to see your ODBC driver vendor for information on installation and configuration.

3. Install your DBMS if needed.

If you do not already have a DBMS installed, you will need to follow the vendor's instructions for installation. Most users will have a DBMS installed and will be working with an established database.

Setting Up a Database

We will assume that the database COFFEEBREAK already exists. (Creating a database is not at all difficult, but it requires special permissions and is normally done by a database administrator.) When you create the tables used as examples in this tutorial, they will be in the default database. We purposely kept the size and number of tables small to keep things manageable.

Suppose that our sample database is being used by the proprietor of a small coffee house called The Coffee Break, where coffee beans are sold by the pound and brewed coffee is sold by the cup. To keep things simple, also suppose that the proprietor needs only two tables, one for types of coffee and one for coffee suppliers.

First we will show you how to open a connection with your DBMS, and then, since what JDBC does is to send your SQL code to your DBMS, we will demonstrate some SQL code. After that, we will show you how easy it is to use JDBC to pass these SQL statements to your DBMS and process the results that are returned.

This code has been tested on most of the major DBMS products. However, you may encounter some compatibility problems using it with older ODBC drivers with the JDBC-ODBC Bridge.

Establishing a Connection

The first thing you need to do is establish a connection with the DBMS you want to use.

This involves two steps: (1) loading the driver and (2) making the connection.

Loading Drivers

Loading the driver or drivers you want to use is very simple and involves just one line of code. If, for example, you want to use the JDBC-ODBC Bridge driver, the following code will load it:

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

Your driver documentation will give you the class name to use. For instance, if the class name is `jdbc.DriverXYZ`, you would load the driver with the following line of code:

```
Class.forName("jdbc.DriverXYZ");
```

You do not need to create an instance of a driver and register it with the `DriverManager` because calling `Class.forName` will do that for you automatically. If you were to create your own instance, you would be creating an unnecessary duplicate, but it would do no harm.

When you have loaded a driver, it is available for making a connection with a DBMS.

Making the Connection

The second step in establishing a connection is to have the appropriate driver connect to the DBMS. The following line of code illustrates the general idea:

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

This step is also simple, with the hardest thing being what to supply for `url`. If you are using the JDBC-ODBC Bridge driver, the JDBC URL will start with `jdbc:odbc:`. The rest of the URL is generally your data source name or database system. So, if you are using ODBC to access an ODBC data source called "Fred", for example, your JDBC URL could be `jdbc:odbc:Fred`. In place of "myLogin" you put the name you use to log in to the DBMS; in place of "myPassword" you put your password for the DBMS. So if you log in to your DBMS with a login name of "Fernanda" and a password of "J8", just these two lines of code will establish a connection:

```
String url = "jdbc:odbc:Fred";  
Connection con = DriverManager.getConnection(url, "Fernanda", "J8");
```

If you are using a JDBC driver developed by a third party, the documentation will tell you what subprotocol to use, that is, what to put after `jdbc:` in the JDBC URL. For example, if the driver developer has registered the name `acme` as the subprotocol, the first and second parts of the JDBC URL will be `jdbc:acme:`. The driver documentation will also give you guidelines for the rest of the JDBC URL. This last part of the JDBC URL supplies information for identifying the data source.

If one of the drivers you loaded recognizes the JDBC URL supplied to the method

DriverManager.getConnection , that driver will establish a connection to the DBMS specified in the JDBC URL. The DriverManager class, true to its name, manages all of the details of establishing the connection for you behind the scenes. Unless you are writing a driver, you will probably never use any of the methods in the interface Driver , and the only DriverManager method you really need to know is DriverManager.getConnection .

The connection returned by the method DriverManager.getConnection is an open connection you can use to create JDBC statements that pass your SQL statements to the DBMS. In the previous example, con is an open connection, and we will use it in the examples that follow.

Setting Up Tables

Creating a Table

First, we will create one of the tables in our example database. This table, COFFEES , contains the essential information about the coffees sold at The Coffee Break, including the coffee names, their prices, the number of pounds sold the current week, and the number of pounds sold to date. The table COFFEES , which we describe in more detail later, is shown here:

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	0	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

The column storing the coffee name is COF_NAME, and it holds values with an SQL type of VARCHAR and a maximum length of 32 characters. Since we will use different names for each type of coffee sold, the name will uniquely identify a particular coffee and can therefore serve as the primary key. The second column, named SUP_ID , will hold a number that identifies the coffee supplier; this number will be of SQL type INTEGER . The third column, called PRICE, stores values with an SQL type of FLOAT because it needs to hold values with decimal points. (Note that money values would normally be stored in an SQL type DECIMAL or NUMERIC , but because of differences among DBMSs and to avoid incompatibility with older versions of JDBC, we are using

the more standard type FLOAT for this tutorial.) The column named SALES stores values of SQL type INTEGER and indicates the number of pounds of coffee sold during the current week. The final column, TOTAL , contains an SQL INTEGER which gives the total number of pounds of coffee sold to date.

SUPPLIERS , the second table in our database, gives information about each of the suppliers:

SUP_ID	SUP_NAME	STREET	CITY	STATE	ZIP
101	Acme, Inc.	99 Market Street	Groundsville	CA	95199
49	Superior Coffee	1 Party Place	Mendocino	CA	95460
150	The High Ground	100 Coffee Lane	Meadows	CA	93966

The tables COFFEES and SUPPLIERS both contain the column SUP_ID , which means that these two tables can be used in SELECT statements to get data based on the information in both tables. The column SUP_ID is the primary key in the table SUPPLIERS , and as such, it uniquely identifies each of the coffee suppliers. In the table COFFEES , SUP_ID is called a foreign key. (You can think of a foreign key as being foreign in the sense that it is imported from another table.) Note that each SUP_ID number appears only once in the SUPPLIERS table; this is required for it to be a primary key. In the COFFEES table, where it is a foreign key, however, it is perfectly all right for there to be duplicate SUP_ID numbers because one supplier may sell many types of coffee. Later in this chapter, you will see an example of how to use primary and foreign keys in a SELECT statement.

The following SQL statement creates the table COFFEES . The entries within the outer pair of parentheses consist of the name of a column followed by a space and the SQL type to be stored in that column. A comma separates the entry for one column (consisting of column name and SQL type) from the next one. The type VARCHAR is created with a maximum length, so it takes a parameter indicating that maximum length. The parameter must be in parentheses following the type. The SQL statement shown here, for example, specifies that the name in column COF_NAME may be up to 32 characters long:

```
CREATE TABLE COFFEES
(COF_NAME VARCHAR(32),
SUP_ID INTEGER,
PRICE FLOAT,
SALES INTEGER,
TOTAL INTEGER)
```

This code does not end with a DBMS statement terminator, which can vary from DBMS to DBMS. For example, Oracle uses a semicolon (;) to indicate the end of a statement,

and Sybase uses the word `go`. The driver you are using will automatically supply the appropriate statement terminator, and you will not need to include it in your JDBC code.

Another thing we should point out about SQL statements is their form. In the `CREATE TABLE` statement, key words are printed in all capital letters, and each item is on a separate line. SQL does not require either; these conventions simply make statements easier to read. The standard in SQL is that keywords are not case sensitive, so, for example, the following `SELECT` statement can be written various ways. As an example, these two versions below are equivalent as far as SQL is concerned:

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Last_Name LIKE "Washington"
select First_Name, Last_Name from Employees where
Last_Name like "Washington"
```

Quoted material, however, is case sensitive: in the name " Washington, " " W " must be capitalized, and the rest of the letters must be lowercase.

Requirements can vary from one DBMS to another when it comes to identifier names. For example, some DBMSs require that column and table names be given exactly as they were created in the `CREATE TABLE` statement, while others do not. To be safe, we will use all uppercase for identifiers such as `COFFEES` and `SUPPLIERS` because that is how we defined them.

So far we have written the SQL statement that creates the table `COFFEES`. Now let's put quotation marks around it (making it a string) and assign that string to the variable `createTableCoffees` so that we can use the variable in our JDBC code later. As just shown, the DBMS does not care about where lines are divided, but in the Java programming language, a `String` object that extends beyond one line will not compile. Consequently, when you are giving strings, you need to enclose each line in quotation marks and use a plus sign (+) to concatenate them:

```
String createTableCoffees = "CREATE TABLE COFFEES " +
"(COF_NAME VARCHAR(32), SUP_ID INTEGER, PRICE FLOAT, " +
"SALES INTEGER, TOTAL INTEGER)";
```

The data types we used in our `CREATE TABLE` statement are the generic SQL types (also called JDBC types) that are defined in the class `java.sql.Types`. DBMSs generally use these standard types, so when the time comes to try out some JDBC applications, you can just use the application `CreateCoffees.java`, which uses the `CREATE TABLE` statement. If your DBMS uses its own local type names, we supply another application for you, which we will explain fully later.

Before running any applications, however, we are going to walk you through the basics of JDBC.

Creating JDBC Statements

A `Statement` object is what sends your SQL statement to the DBMS. You simply create a `Statement` object and then execute it, supplying the appropriate execute method with the SQL statement you want to send. For a `SELECT` statement, the method to use is

`executeQuery` . For statements that create or modify tables, the method to use is `executeUpdate` .

It takes an instance of an active connection to create a Statement object. In the following example, we use our Connection object `con` to create the Statement object `stmt` :

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

At this point `stmt` exists, but it does not have an SQL statement to pass on to the DBMS. We need to supply that to the method we use to execute `stmt` . For example, in the following code fragment, we supply `executeUpdate` with the SQL statement from the example above:

```
stmt.executeUpdate("CREATE TABLE COFFEES " +  
    "(COF_NAME VARCHAR(32), SUP_ID INTEGER, PRICE FLOAT, " +  
    "SALES INTEGER, TOTAL INTEGER)");
```

Since we made a string out of the SQL statement and assigned it to the variable `createTableCoffees` , we could have written the code in this alternate form:

```
stmt.executeUpdate(createTableCoffees);
```

Executing Statements

We used the method `executeUpdate` because the SQL statement contained in `createTableCoffees` is a DDL (data definition language) statement. Statements that create a table, alter a table, or drop a table are all examples of DDL statements and are executed with the method `executeUpdate` . As you might expect from its name, the method `executeUpdate` is also used to execute SQL statements that update a table. In practice, `executeUpdate` is used far more often to update tables than it is to create them because a table is created once but may be updated many times.

The method used most often for executing SQL statements is `executeQuery` . This method is used to execute `SELECT` statements, which comprise the vast majority of SQL statements. You will see how to use this method shortly.

Entering Data into a Table

We have shown how to create the table `COFFEES` by specifying the names of the columns and the data types to be stored in those columns, but this only sets up the structure of the table. The table does not yet contain any data. We will enter our data into the table one row at a time, supplying the information to be stored in each column of that row. Note that the values to be inserted into the columns are listed in the same order that the columns were declared when the table was created, which is the default order.

The following code inserts one row of data, with Colombian in the column `COF_NAME` , 101 in `SUP_ID` , 7.99 in `PRICE` , 0 in `SALES` , and 0 in `TOTAL` . (Since The Coffee Break has just started out, the amount sold during the week and the total to date are zero for all the coffees to start with.) Just as we did in the code that created the table `COFFEES` , we will create a Statement object and then execute it using the method `executeUpdate` .

Since the SQL statement will not quite fit on one line on the page, we have split it into

two strings concatenated by a plus sign (+) so that it will compile. Pay special attention to the need for a space between COFFEES and VALUES . This space must be within the quotation marks and may be after COFFEES or before VALUES ; without a space, the SQL statement will erroneously be read as " INSERT INTO COFFEESVALUES . . ." and the DBMS will look for the table COFFEESVALUES . Also note that we use single quotation marks around the coffee name because it is nested within double quotation marks. For most DBMSs, the general rule is to alternate double quotation marks and single quotation marks to indicate nesting.

```
Statement stmt = con.createStatement();
stmt.executeUpdate(
    "INSERT INTO COFFEES " +
    "VALUES ('Colombian', 101, 7.99, 0, 0)");
```

The code that follows inserts a second row into the table COFFEES . Note that we can just reuse the Statement object stmt rather than having to create a new one for each execution.

```
stmt.executeUpdate("INSERT INTO COFFEES " +
    "VALUES ('French_Roast', 49, 8.99, 0, 0)");
```

Values for the remaining rows can be inserted as follows:

```
stmt.executeUpdate("INSERT INTO COFFEES " +
    "VALUES ('Espresso', 150, 9.99, 0, 0)");
stmt.executeUpdate("INSERT INTO COFFEES " +
    "VALUES ('Colombian_Decaf', 101, 8.99, 0, 0)");
stmt.executeUpdate("INSERT INTO COFFEES " +
    "VALUES ('French_Roast_Decaf', 49, 9.99, 0, 0)");
```

Getting Data from a Table

Now that the table COFFEES has values in it, we can write a SELECT statement to access those values. The star (*) in the following SQL statement indicates that all columns should be selected. Since there is no WHERE clause to narrow down the rows from which to select, the following SQL statement selects the whole table:

```
SELECT * FROM COFFEES
```

The result, which is the entire table, will look similar to the following:

COF_NAME	SUP_ID	PRICE	SALES	TOTAL	
Colombian	101	7.99	0	0	
French_Roast	49	8.99	0	0	
Espresso	150	9.99	0	0	
Colombian_Decaf		101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0	

The result above is what you would see on your terminal if you entered the SQL query directly to the database system. When we access a database through a Java application, as we will be doing shortly, we will need to retrieve the results so that we can use them. You will see how to do this in the next section.

Here is another example of a SELECT statement; this one will get a list of coffees and their respective prices per pound:

```
SELECT COF_NAME, PRICE FROM COFFEES
```

The results of this query will 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

The SELECT statement above generates the names and prices of all of the coffees in the table. The following SQL statement limits the coffees selected to just those that cost less than \$9.00 per pound:

```
SELECT COF_NAME, PRICE
FROM COFFEES
WHERE PRICE < 9.00
```

The results would look similar to this:

COF_NAME	PRICE
Colombian	7.99
French_Roast	8.99
Colombian Decaf	8.99

Updating Tables

Suppose that after a successful first week, the proprietor of The Coffee Break wants to update the SALES column in the table COFFEES by entering the number of pounds sold for each type of coffee. The SQL statement to update one row might look like this:

```
String updateString = "UPDATE COFFEES " +
    "SET SALES = 75 " +
    "WHERE COF_NAME LIKE 'Colombian'";
```

Using the Statement object stmt , this JDBC code executes the SQL statement contained in updateString :

```
stmt.executeUpdate(updateString);
```

The table COFFEES will now look like this:

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	75	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

Note that we have not yet updated the column TOTAL , so it still has the value 0 .

Now let's select the row we updated, retrieve the values in the columns COF_NAME and SALES , and print out those values:

```
String query = "SELECT COF_NAME, SALES FROM COFFEES " +
    "WHERE COF_NAME LIKE 'Colombian'";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
    String s = rs.getString("COF_NAME");
    int n = rs.getInt("SALES");
    System.out.println(n + " pounds of " + s +
        " sold this week.");
}
```

This will print the following:

```
75 pounds of Colombian sold this week.
```

Since the WHERE clause limited the selection to only one row, there was just one row in the ResultSet rs and one line printed as output. Accordingly, it is possible to write the code without a while loop:

```
rs.next();
String s = rs.getString(1);
int n = rs.getInt(2);
System.out.println(n + " pounds of " + s + " sold this week.");
```

Even when there is only one row in a result set, you need to use the method next to access it. A ResultSet object is created with a cursor pointing above the first row. The first call to the next method positions the cursor on the first (and in this case, only) row of rs . In this code, next is called only once, so if there happened to be another row, it would never be accessed.

Now let's update the TOTAL column by adding the weekly amount sold to the existing total, and then let's print out the number of pounds sold to date:

```
String updateString = "UPDATE COFFEES " +
    "SET TOTAL = TOTAL + 75 " +
    "WHERE COF_NAME LIKE 'Colombian'";
stmt.executeUpdate(updateString);
String query = "SELECT COF_NAME, TOTAL FROM COFFEES " +
    "WHERE COF_NAME LIKE 'Colombian'";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
    String s = rs.getString(1);
    int n = rs.getInt(2);
    System.out.println(n + " pounds of " + s + " sold to date.");
}
```

Note that in this example, we used the column index instead of the column name, supplying the index 1 to getString (the first column of the result set is COF_NAME), and the index 2 to getInt (the second column of the result set is TOTAL). It is important to distinguish between a column's index in the database table as opposed to its index in the result set table. For example, TOTAL is the fifth column in the table COFFEES but the second column in the result set generated by the query in the example above.

Milestone: The Basics of JDBC

You have just reached a milestone.

With what we have done so far, you have learned the basics of JDBC. You have seen how to create a table, insert values into it, query the table, retrieve results, and update the table. These are the nuts and bolts of using a database, and you can now utilize them in a program written in the Java programming language using the JDBC 1.0 API. We have used only very simple queries in our examples so far, but as long as the driver and DBMS support them, you can send very complicated SQL queries using only the basic JDBC API we have covered so far.

The rest of this lesson looks at how to use features that are a little more advanced: prepared statements, stored procedures, and transactions. It also illustrates warnings and exceptions and gives an example of how to convert a JDBC application into an applet. The final part of this lesson is sample code that you can run yourself.

Using Prepared Statements

Sometimes it is more convenient or more efficient to use a `PreparedStatement` object for sending SQL statements to the database. This special type of statement is derived from the more general class, `Statement`, that you already know.

When to Use a PreparedStatement Object

If you want to execute a `Statement` object many times, it will normally reduce execution time to use a `PreparedStatement` object instead.

The main feature of a `PreparedStatement` object is that, unlike a `Statement` object, it is given an SQL statement when it is created. The advantage to this is that in most cases, this SQL statement will be sent to the DBMS right away, where it will be compiled. As a result, the `PreparedStatement` object contains not just an SQL statement, but an SQL statement that has been precompiled. This means that when the `PreparedStatement` is executed, the DBMS can just run the `PreparedStatement`'s SQL statement without having to compile it first.

Although `PreparedStatement` objects can be used for SQL statements with no parameters, you will probably use them most often for SQL statements that take parameters. The advantage of using SQL statements that take parameters is that you can use the same statement and supply it with different values each time you execute it. You will see an example of this in the following sections.

Creating a PreparedStatement Object

As with `Statement` objects, you create `PreparedStatement` objects with a `Connection`

method. Using our open connection `con` from previous examples, you might write code such as the following to create a `PreparedStatement` object that takes two input parameters:

```
PreparedStatement updateSales = con.prepareStatement(
    "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
```

The variable `updateSales` now contains the SQL statement, "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?" , which has also, in most cases, been sent to the DBMS and been precompiled.

Supplying Values for PreparedStatement Parameters

You will need to supply values to be used in place of the question mark placeholders, if there are any, before you can execute a `PreparedStatement` object. You do this by calling one of the `setXXX` methods defined in the class `PreparedStatement` . If the value you want to substitute for a question mark is a Java `int` , you call the method `setInt`. If the value you want to substitute for a question mark is a Java `String` , you call the method `setString` , and so on. In general, there is a `setXXX` method for each type in the Java programming language.

Using the `PreparedStatement` object `updateSales` from the previous example, the following line of code sets the first question mark placeholder to a Java `int` with a value of 75:

```
updateSales.setInt(1, 75);
```

As you might surmise from the example, the first argument given to a `setXXX` method indicates which question mark placeholder is to be set, and the second argument indicates the value to which it is to be set. The next example sets the second placeholder parameter to the string " Colombian ":

```
updateSales.setString(2, "Colombian");
```

After these values have been set for its two input parameters, the SQL statement in `updateSales` will be equivalent to the SQL statement in the `String` object `updateString` that we used in the previous update example. Therefore, the following two code fragments accomplish the same thing:

Code Fragment 1:

```
String updateString = "UPDATE COFFEES SET SALES = 75 " +
    "WHERE COF_NAME LIKE 'Colombian'";
stmt.executeUpdate(updateString);
```

Code Fragment 2:

```
PreparedStatement updateSales = con.prepareStatement(
    "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
updateSales.setInt(1, 75);
updateSales.setString(2, "Colombian");
updateSales.executeUpdate();
```

We used the method `executeUpdate` to execute both the `Statement` `stmt` and the

PreparedStatement updateSales . Notice, however, that no argument is supplied to executeUpdate when it is used to execute updateSales . This is true because updateSales already contains the SQL statement to be executed.

Looking at these examples, you might wonder why you would choose to use a PreparedStatement object with parameters instead of just a simple statement, since the simple statement involves fewer steps. If you were going to update the SALES column only once or twice, then there would be no need to use an SQL statement with input parameters. If you will be updating often, on the other hand, it might be much easier to use a PreparedStatement object, especially in situations where you can use a for loop or while loop to set a parameter to a succession of values. You will see an example of this later in this section.

Once a parameter has been set with a value, it will retain that value until it is reset to another value or the method clearParameters is called. Using the PreparedStatement object updateSales , the following code fragment illustrates reusing a prepared statement after resetting the value of one of its parameters and leaving the other one the same:

```
updateSales.setInt(1, 100);
updateSales.setString(2, "French_Roast");
updateSales.executeUpdate();
// changes SALES column of French Roast row to 100
updateSales.setString(2, "Espresso");
updateSales.executeUpdate();
// changes SALES column of Espresso row to 100 (the first
// parameter stayed 100, and the second parameter was reset
// to "Espresso")
```

Using a Loop to Set Values

You can often make coding easier by using a for loop or a while loop to set values for input parameters.

The code fragment that follows demonstrates using a for loop to set values for parameters in the PreparedStatement object updateSales . The array salesForWeek holds the weekly sales amounts. These sales amounts correspond to the coffee names listed in the array coffees , so that the first amount in salesForWeek (175) applies to the first coffee name in coffees (" Colombian "), the second amount in salesForWeek (150) applies to the second coffee name in coffees (" French_Roast "), and so on. This code fragment demonstrates updating the SALES column for all the coffees in the table COFFEES :

```
PreparedStatement updateSales;
String updateString = "update COFFEES " +
    "set SALES = ? where COF_NAME like ?";
updateSales = con.prepareStatement(updateString);
int [] salesForWeek = { 175, 150, 60, 155, 90 };
String [] coffees = {"Colombian", "French_Roast", "Espresso",
    "Colombian_Decaf", "French_Roast_Decaf"};
int len = coffees.length;
for(int i = 0; i < len; i++) {
```

```

        updateSales.setInt(1, salesForWeek[i]);
        updateSales.setString(2, coffees[i]);
        updateSales.executeUpdate();
    }

```

When the proprietor wants to update the sales amounts for the next week, he can use this same code as a template. All he has to do is enter the new sales amounts in the proper order in the array `salesForWeek` . The coffee names in the array `coffees` remain constant, so they do not need to be changed. (In a real application, the values would probably be input from the user rather than from an initialized Java array.)

Return Values for the Method `executeUpdate`

Whereas `executeQuery` returns a `ResultSet` object containing the results of the query sent to the DBMS, the return value for `executeUpdate` is an `int` that indicates how many rows of a table were updated. For instance, the following code shows the return value of `executeUpdate` being assigned to the variable `n` :

```

updateSales.setInt(1, 50);
updateSales.setString(2, "Espresso");
int n = updateSales.executeUpdate();
// n = 1 because one row had a change in it

```

The table `COFFEES` was updated by having the value 50 replace the value in the column `SALES` in the row for Espresso . That update affected one row in the table, so `n` is equal to 1 .

When the method `executeUpdate` is used to execute a DDL statement, such as in creating a table, it returns the `int 0` . Consequently, in the following code fragment, which executes the DDL statement used to create the table `COFFEES` , `n` will be assigned a value of 0 :

```

int n = executeUpdate(createTableCoffees); // n = 0

```

Note that when the return value for `executeUpdate` is 0 , it can mean one of two things: (1) the statement executed was an update statement that affected zero rows, or (2) the statement executed was a DDL statement.

Using Joins

Sometimes you need to use two or more tables to get the data you want. For example, suppose the proprietor of The Coffee Break wants a list of the coffees he buys from Acme, Inc. This involves information in the `COFFEES` table as well as the yet-to-be-created `SUPPLIERS` table. This is a case where a join is needed. A join is a database operation that relates two or more tables by means of values that they share in common. In our example database, the tables `COFFEES` and `SUPPLIERS` both have the column `SUP_ID` , which can be used to join them.

Before we go any further, we need to create the table SUPPLIERS and populate it with values.

The code below creates the table SUPPLIERS :

```
String createSUPPLIERS = "create table SUPPLIERS " +
    "(SUP_ID INTEGER, SUP_NAME VARCHAR(40), " +
    "STREET VARCHAR(40), CITY VARCHAR(20), " +
    "STATE CHAR(2), ZIP CHAR(5))";
stmt.executeUpdate(createSUPPLIERS);
```

The following code inserts rows for three suppliers into SUPPLIERS :

```
stmt.executeUpdate("insert into SUPPLIERS values (101, " +
    "'Acme, Inc.', '99 Market Street', 'Groundsville', " + "'CA', '95199'");
stmt.executeUpdate("insert into SUPPLIERS values (49, " +
    "'Superior Coffee', '1 Party Place', 'Mendocino', 'CA', " + "'95460'");
stmt.executeUpdate("insert into SUPPLIERS values (150, " +
    "'The High Ground', '100 Coffee Lane', 'Meadows', 'CA', " + "'93966'");
```

The following code selects the whole table and lets us see what the table SUPPLIERS looks like:

```
ResultSet rs = stmt.executeQuery("select * from SUPPLIERS");
```

The result set will look similar to this:

SUP_ID	SUP_NAME	STREET	CITY	STATE	ZIP
101	Acme, Inc.	99 Market Street	Groundsville	CA	95199
49	Superior Coffee	1 Party Place	Mendocino	CA	95460
150	The High Ground	100 Coffee Lane	Meadows	CA	93966

Now that we have the tables COFFEES and SUPPLIERS , we can proceed with the scenario where the owner wants to get a list of the coffees he buys from a particular supplier. The names of the suppliers are in the table SUPPLIERS , and the names of the coffees are in the table COFFEES . Since both tables have the column SUP_ID , this column can be used in a join. It follows that you need some way to distinguish which SUP_ID column you are referring to. This is done by preceding the column name with the table name, as in " COFFEES.SUP_ID " to indicate that you mean the column SUP_ID in the table COFFEES . The following code, in which stmt is a Statement object, will select the coffees bought from Acme, Inc.:

```
String query = "
SELECT COFFEES.COF_NAME " +
    "FROM COFFEES, SUPPLIERS " +
    "WHERE SUPPLIERS.SUP_NAME LIKE 'Acme, Inc.' " +
    "and SUPPLIERS.SUP_ID = COFFEES.SUP_ID";

ResultSet rs = stmt.executeQuery(query);
```

```
System.out.println("Coffees bought from Acme, Inc.: ");
while (rs.next()) {
    String coffeeName = rs.getString("COF_NAME");
    System.out.println("    " + coffeeName);
}
```

This will produce the following output:

```
Coffees bought from Acme, Inc.:
Colombian
Colombian_Decaf
```

Using Transactions

There are times when you do not want one statement to take effect unless another one also succeeds. For example, when the proprietor of The Coffee Break updates the amount of coffee sold each week, he will also want to update the total amount sold to date. However, he will not want to update one without also updating the other; otherwise, the data will be inconsistent. The way to be sure that either both actions occur or neither action occurs is to use a transaction. A transaction is a set of one or more statements that are executed together as a unit, so either all of the statements are executed, or none of the statements is executed.

Disabling Auto-commit Mode

When a connection is created, it is in auto-commit mode. This means that each individual SQL statement is treated as a transaction and will be automatically committed right after it is executed. (To be more precise, the default is for an SQL statement to be committed when it is completed, not when it is executed. A statement is completed when all of its result sets and update counts have been retrieved. In almost all cases, however, a statement is completed, and therefore committed, right after it is executed.)

The way to allow two or more statements to be grouped into a transaction is to disable auto-commit mode. This is demonstrated in the following line of code, where `con` is an active connection:

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

Committing a Transaction

Once auto-commit mode is disabled, no SQL statements will be committed until you call the method `commit` explicitly. All statements executed after the previous call to the method `commit` will be included in the current transaction and will be committed together as a unit. The following code, in which `con` is an active connection, illustrates a transaction:

```
con.setAutoCommit(false);
PreparedStatement updateSales = con.prepareStatement(
    "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
updateSales.setInt(1, 50);
```

```

updateSales.setString(2, "Colombian");
updateSales.executeUpdate();
PreparedStatement updateTotal = con.prepareStatement(
    "UPDATE COFFEES SET TOTAL = TOTAL + ? WHERE COF_NAME LIKE ?");
updateTotal.setInt(1, 50);
updateTotal.setString(2, "Colombian");
updateTotal.executeUpdate();
con.commit();
con.setAutoCommit(true);

```

In this example, auto-commit mode is disabled for the connection `con`, which means that the two prepared statements `updateSales` and `updateTotal` will be committed together when the method `commit` is called. Whenever the `commit` method is called (either automatically when auto-commit mode is enabled or explicitly when it is disabled), all changes resulting from statements in the transaction will be made permanent. In this case, that means that the `SALES` and `TOTAL` columns for Colombian coffee have been changed to 50 (if `TOTAL` had been 0 previously) and will retain this value until they are changed with another update statement. [TransactionPairs.java](#) illustrates a similar kind of transaction but uses a for loop to supply values to the `setXXX` methods for `updateSales` and `updateTotal`.

The final line of the previous example enables auto-commit mode, which means that each statement will once again be committed automatically when it is completed. You will then be back to the default state where you do not have to call the method `commit` yourself. It is advisable to disable auto-commit mode only while you want to be in transaction mode. This way, you avoid holding database locks for multiple statements, which increases the likelihood of conflicts with other users.

Using Transactions to Preserve Data Integrity

In addition to grouping statements together for execution as a unit, transactions can help to preserve the integrity of the data in a table. For instance, suppose that an employee was supposed to enter new coffee prices in the table `COFFEES` but delayed doing it for a few days. In the meantime, prices rose, and today the owner is in the process of entering the higher prices. The employee finally gets around to entering the now outdated prices at the same time that the owner is trying to update the table. After inserting the outdated prices, the employee realizes that they are no longer valid and calls the `Connection` method `rollback` to undo their effects. (The method `rollback` aborts a transaction and restores values to what they were before the attempted update.) At the same time, the owner is executing a `SELECT` statement and printing out the new prices. In this situation, it is possible that the owner will print a price that was later rolled back to its previous value, making the printed price incorrect.

This kind of situation can be avoided by using transactions. If a DBMS supports transactions, and almost all of them do, it will provide some level of protection against conflicts that can arise when two users access data at the same time.

To avoid conflicts during a transaction, a DBMS will use locks, mechanisms for blocking

access by others to the data that is being accessed by the transaction. (Note that in auto-commit mode, where each statement is a transaction, locks are held for only one statement.) Once a lock is set, it will remain in force until the transaction is committed or rolled back. For example, a DBMS could lock a row of a table until updates to it have been committed. The effect of this lock would be to prevent a user from getting a dirty read, that is, reading a value before it is made permanent. (Accessing an updated value that has not been committed is considered a dirty read because it is possible for that value to be rolled back to its previous value. If you read a value that is later rolled back, you will have read an invalid value.)

How locks are set is determined by what is called a transaction isolation level, which can range from not supporting transactions at all to supporting transactions that enforce very strict access rules.

One example of a transaction isolation level is `TRANSACTION_READ_COMMITTED`, which will not allow a value to be accessed until after it has been committed. In other words, if the transaction isolation level is set to `TRANSACTION_READ_COMMITTED`, the DBMS will not allow dirty reads to occur. The interface `Connection` includes five values which represent the transaction isolation levels you can use in JDBC.

Normally, you do not need to do anything about the transaction isolation level; you can just use the default one for your DBMS. JDBC allows you to find out what transaction isolation level your DBMS is set to (using the `Connection` method `getTransactionIsolation`) and also allows you to set it to another level (using the `Connection` method `setTransactionIsolation`). Keep in mind, however, that even though JDBC allows you to set a transaction isolation level, doing so will have no effect unless the driver and DBMS you are using support it.

When to Call the Method `rollback`

As mentioned earlier, calling the method `rollback` aborts a transaction and returns any values that were modified to their previous values. If you are trying to execute one or more statements in a transaction and get an `SQLException`, you should call the method `rollback` to abort the transaction and start the transaction all over again. That is the only way to be sure of what has been committed and what has not been committed. Catching an `SQLException` tells you that something is wrong, but it does not tell you what was or was not committed. Since you cannot count on the fact that nothing was committed, calling the method `rollback` is the only way to be sure.

[TransactionPairs.java](#) demonstrates a transaction and includes a catch block that invokes the method `rollback`. In this particular situation, it is not really necessary to call `rollback`, and we do it mainly to illustrate how it is done. If the application continued and used the results of the transaction, however, it would be necessary to include a call to `rollback` in the catch block in order to protect against using possibly incorrect data.

Stored Procedures

A stored procedure is a group of SQL statements that form a logical unit and perform a particular task. Stored procedures are used to encapsulate a set of operations or queries to execute on a database server. For example, operations on an employee database (hire, fire, promote, lookup) could be coded as stored procedures executed by application code. Stored procedures can be compiled and executed with different parameters and results, and they may have any combination of input, output, and input/output parameters.

Stored procedures are supported by most DBMSs, but there is a fair amount of variation in their syntax and capabilities. For this reason, we will show you a simple example of what a stored procedure looks like and how it is invoked from JDBC, but this sample is not intended to be run.

SQL Statements for Creating a Stored Procedure

This section looks at a very simple stored procedure that has no parameters. Even though most stored procedures do something more complex than this example, it serves to illustrate some basic points about them. As previously stated, the syntax for defining a stored procedure is different for each DBMS. For example, some use `begin . . . end` or other keywords to indicate the beginning and ending of the procedure definition. In some DBMSs, the following SQL statement creates a stored procedure:

```
create procedure SHOW_SUPPLIERS
as
select SUPPLIERS.SUP_NAME, COFFEES.COF_NAME
from SUPPLIERS, COFFEES
where SUPPLIERS.SUP_ID = COFFEES.SUP_ID
order by SUP_NAME
```

The following code puts the SQL statement into a string and assigns it to the variable `createProcedure`, which we will use later:

```
String createProcedure = "create procedure SHOW_SUPPLIERS " +
    "as " +
    "select SUPPLIERS.SUP_NAME, COFFEES.COF_NAME " +
    "from SUPPLIERS, COFFEES " +
    "where SUPPLIERS.SUP_ID = COFFEES.SUP_ID " +
    "order by SUP_NAME";
```

The following code fragment uses the `Connection` object `con` to create a `Statement` object, which is used to send the SQL statement creating the stored procedure to the database:

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

The procedure `SHOW_SUPPLIERS` will be compiled and stored in the database as a database object that can be called, similar to the way you would call a method.

Calling a Stored Procedure from JDBC

JDBC allows you to call a database stored procedure from an application written in the Java programming language. The first step is to create a `CallableStatement` object. As with `Statement` and `PreparedStatement` objects, this is done with an open `Connection` object. A `CallableStatement` object contains a call to a stored procedure; it does not contain the stored procedure itself. The first line of code below creates a call to the stored procedure `SHOW_SUPPLIERS` using the connection `con`. The part that is enclosed in curly braces is the escape syntax for stored procedures. When the driver encounters "`{call SHOW_SUPPLIERS}`", it will translate this escape syntax into the native SQL used by the database to call the stored procedure named `SHOW_SUPPLIERS`.

```
CallableStatement cs = con.prepareCall("{call SHOW_SUPPLIERS}");
ResultSet rs = cs.executeQuery();
```

The `ResultSet` `rs` will be similar to the following:

SUP_NAME	COF_NAME
Acme, Inc.	Colombian
Acme, Inc.	Colombian_Decaf
Superior Coffee	French_Roast
Superior Coffee	French_Roast_Decaf
The High Ground	Espresso

Note that the method used to execute `cs` is `executeQuery` because `cs` calls a stored procedure that contains one query and thus produces one result set. If the procedure had contained one update or one DDL statement, the method `executeUpdate` would have been the one to use. It is sometimes the case, however, that a stored procedure contains more than one SQL statement, in which case it will produce more than one result set, more than one update count, or some combination of result sets and update counts. In this case, where there are multiple results, the method `execute` should be used to execute the `CallableStatement`.

The class `CallableStatement` is a subclass of `PreparedStatement`, so a `CallableStatement` object can take input parameters just as a `PreparedStatement` object can. In addition, a `CallableStatement` object can take output parameters or parameters that are for both input and output. `INOUT` parameters and the method `execute` are used rarely. For complete information, refer to `JDBC Database Access with Java`.

Creating Complete JDBC Applications

Up to this point, you have seen only code fragments. Later in this section you will see sample programs that are complete applications you can run.

The first sample code creates the table `COFFEES`; the second one inserts values into the table and prints the results of a query. The third application creates the table `SUPPLIERS`, and the fourth populates it with values. After you have run this code, you can try a query that is a join between the tables `COFFEES` and `SUPPLIERS`, as in the fifth code example. The sixth code sample is an application that demonstrates a

transaction and also shows how to set placeholder parameters in a PreparedStatement object using a for loop.

Because they are complete applications, they include some elements of the Java programming language we have not shown before in the code fragments. We will explain these elements briefly here.

Putting Code in a Class Definition

In the Java programming language, any code you want to execute must be inside a class definition. You type the class definition in a file and give the file the name of the class with .java appended to it. So if you have a class named MySQLStatement, its definition should be in a file named MySQLStatement.java .

Importing Classes to Make Them Visible

The first thing to do is to import the packages or classes you will be using in the new class. The classes in our examples all use the java.sql package (the JDBC API), which is made available when the following line of code precedes the class definition:

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

The star (*) indicates that all of the classes in the package java.sql are to be imported. Importing a class makes it visible and means that you do not have to write out the fully qualified name when you use a method or field from that class. If you do not include " import java.sql.*; " in your code, you will have to write " java.sql. " plus the class name in front of all the JDBC fields or methods you use every time you use them. Note that you can import individual classes selectively rather than a whole package. Java does not require that you import classes or packages, but doing so makes writing code a lot more convenient.

Any lines importing classes appear at the top of all the code samples, as they must if they are going to make the imported classes visible to the class being defined. The actual class definition follows any lines that import classes.

Using the main Method

If a class is to be executed, it must contain a static public main method. This method comes right after the line declaring the class and invokes the other methods in the class. The keyword static indicates that this method operates on a class level rather than on individual instances of a class. The keyword public means that members of any class can access this method. Since we are not just defining classes to be used by other classes but instead want to run them, the example applications in this chapter all include a main method.

Using try and catch Blocks

Something else all the sample applications include is try and catch blocks. These are the Java programming language's mechanism for handling exceptions. Java requires that when a method throws an exception, there be some mechanism to handle it. Generally a

catch block will catch the exception and specify what happens (which you may choose to be nothing). In the sample code, we use two try blocks and two catch blocks. The first try block contains the method `Class.forName`, from the `java.lang` package. This method throws a `ClassNotFoundException`, so the catch block immediately following it deals with that exception. The second try block contains JDBC methods, which all throw `SQLExceptions`, so one catch block at the end of the application can handle all of the rest of the exceptions that might be thrown because they will all be `SQLException` objects.

Retrieving Exceptions

JDBC lets you see the warnings and exceptions generated by your DBMS and by the Java compiler. To see exceptions, you can have a catch block print them out. For example, the following two catch blocks from the sample code print out a message explaining the exception:

```
try {
    // Code that could generate an exception goes here.
    // If an exception is generated, the catch block below
    // will print out information about it.
} catch(SQLException ex) {
    System.err.println("SQLException: " + ex.getMessage());
}

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

If you were to run `CreateCOFFEES.java` twice, you would get an error message similar to this:

```
SQLException: There is already an object named 'COFFEES'
in the database.
Severity 16, State 1, Line 1
```

This example illustrates printing out the message component of an `SQLException` object, which is sufficient for most situations.

There are actually three components, however, and to be complete, you can print them all out. The following code fragment shows a catch block that is complete in two ways. First, it prints out all three parts of an `SQLException` object: the message (a string that describes the error), the SQL state (a string identifying the error according to the X/Open SQLState conventions), and the vendor error code (a number that is the driver vendor's error code number). The `SQLException` object `ex` is caught, and its three components are accessed with the methods `getMessage`, `getSQLState`, and `getErrorCode`.

The second way the following catch block is complete is that it gets all of the exceptions that might have been thrown. If there is a second exception, it will be chained to `ex`, so `ex.getNextException` is called to see if there is another exception. If there is, the while loop continues and prints out the next exception's message, SQLState, and vendor error code. This continues until there are no more exceptions.

```

try {
    // Code that could generate an exception goes here.
    // If an exception is generated, the catch block below
    // will print out information about it.
} catch(SQLException ex) {
    System.out.println("\n--- SQLException caught ---\n");
    while (ex != null) {
        System.out.println("Message: "
            + ex.getMessage ());
        System.out.println("SQLState: "
            + ex.getSQLState ());
        System.out.println("ErrorCode: "
            + ex.getErrorCode ());
        ex = ex.getNextException();
        System.out.println("");
    }
}

```

If you were to substitute the catch block above into [CreateCoffees.java](#) and run it after the table COFFEES had already been created, you would get the following printout:

```

--- SQLException caught ---
Message: There is already an object named 'COFFEES' in the database.
Severity 16, State 1, Line 1
SQLState: 42501
ErrorCode: 2714

```

SQLState is a code defined in X/Open and ANSI-92 that identifies the exception. Two examples of SQLState code numbers and their meanings follow:

```

08001 -- No suitable driver
HY011 -- Operation invalid at this time

```

The vendor error code is specific to each driver, so you need to check your driver documentation for a list of error codes and what they mean.

Retrieving Warnings

SQLWarning objects are a subclass of SQLException that deal with database access warnings. Warnings do not stop the execution of an application, as exceptions do; they simply alert the user that something did not happen as planned. For example, a warning might let you know that a privilege you attempted to revoke was not revoked. Or a warning might tell you that an error occurred during a requested disconnection.

A warning can be reported on a Connection object, a Statement object (including PreparedStatement and CallableStatement objects), or a ResultSet object. Each of these classes has a getWarnings method, which you must invoke in order to see the first warning reported on the calling object. If getWarnings returns a warning, you can call the SQLWarning method getNextWarning on it to get any additional warnings. Executing a statement automatically clears the warnings from a previous statement, so they do not build up. This means, however, that if you want to retrieve warnings reported on a statement, you must do so before you execute another statement.

The following code fragment illustrates how to get complete information about any warnings reported on the Statement object stmt and also on the ResultSet object rs :

```

Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("select COF_NAME from COFFEES");
while (rs.next()) {
    String coffeeName = rs.getString("COF_NAME");
    System.out.println("Coffees available at the Coffee Break: ");
    System.out.println("    " + coffeeName);
    SQLWarning warning = stmt.getWarnings();
    if (warning != null) {
        System.out.println("\n---Warning---\n");
        while (warning != null) {
            System.out.println("Message: "
                + warning.getMessage());
            System.out.println("SQLState: "
                + warning.getSQLState());
            System.out.print("Vendor error code: ");
            System.out.println(warning.getErrorCode());
            System.out.println("");
            warning = warning.getNextWarning();
        }
    }
    SQLWarning warn = rs.getWarnings();
    if (warn != null) {
        System.out.println("\n---Warning---\n");
        while (warn != null) {
            System.out.println("Message: "
                + warn.getMessage());
            System.out.println("SQLState: "
                + warn.getSQLState());
            System.out.print("Vendor error code: ");
            System.out.println(warn.getErrorCode());
            System.out.println("");
            warn = warn.getNextWarning();
        }
    }
}
}

```

Warnings are actually rather uncommon. Of those that are reported, by far the most common warning is a DataTruncation warning, a subclass of SQLWarning. All DataTruncation objects have an SQLState of 01004, indicating that there was a problem with reading or writing data. DataTruncation methods let you find out in which column or parameter data was truncated, whether the truncation was on a read or write operation, how many bytes should have been transferred, and how many bytes were actually transferred.

Running the Sample Applications

You are now ready to actually try out some sample code. The directory `book.html` contains complete, runnable applications that illustrate concepts presented in this chapter and the next. You can download this sample code from the JDBC web site located at:

<http://www.javasoft.com/products/jdbc/book.html> 

Before you can run one of these applications, you will need to edit the file by substituting the appropriate information for the following variables:

url the JDBC URL; parts one and two are supplied by your driver, and the third part specifies your data source **myLogin** your login name or user name **myPassword** your password for the DBMS **myDriver.ClassName** the class name supplied with your driver

The first example application is the class `CreateCoffees` , which is in a file named [CreateCoffees.java](#). Below are instructions for running `CreateCoffees.java` on the three major platforms.

The first line in the instructions below compiles the code in the file `CreateCoffees.java` . If the compilation is successful, it will produce a file named `CreateCoffees.class` , which contains the bytecodes translated from the file `CreateCoffees.java` . These bytecodes will be interpreted by the Java Virtual Machine, which is what makes it possible for Java code to run on any machine with a Java Virtual Machine installed on it.

The second line of code is what actually makes the code run. Note that you use the name of the class, `CreateCoffees` , not the name of the file, `CreateCoffees.class` .

UNIX

```
javac CreateCoffees.java
java CreateCoffees
```

Windows 95/NT

```
javac CreateCoffees.java
java CreateCoffees
```

Creating an Applet from an Application

Suppose that the owner of The Coffee Break wants to display his current coffee prices in an applet on his web page. He can be sure of always displaying the most current price by having the applet get the price directly from his database.

In order to do this, he needs to create two files of code, one with applet code, and one with HTML code. The applet code contains the JDBC code that would appear in a regular application plus additional code for running the applet and displaying the results of the database query. In our example, the applet code is in the file [OutputApplet.java](#). To display our applet in an HTML page, the file [OutputApplet.html](#) tells the browser what to display and where to display it.

The rest of this section will tell you about various elements found in applet code that are not present in standalone application code. Some of these elements involve advanced aspects of the Java programming language. We will give you some rationale and some basic explanation, but explaining them fully is beyond the scope of this tutorial. For purposes of this sample applet, you only need to grasp the general idea, so don't worry if you don't understand everything. You can use the applet code as a template, substituting your own queries for the one in the applet.

Writing Applet Code

To begin with, applets will import classes not used by standalone applications. Our applet

imports two classes that are special to applets: the class `Applet`, which is part of the `java.applet` package, and the class `Graphics`, which is part of the `java.awt` package. This applet also imports the general-purpose class `java.util.Vector` so that we have access to an array-like container whose size can be modified. This code uses `Vector` objects to store query results so that they can be displayed later.

All applets extend the `Applet` class; that is, they are subclasses of `Applet`. Therefore, every applet definition must contain the words `extends Applet`, as shown here:

```
public class MyAppletName extends Applet {  
    ...  
}
```

In our applet example, [OutputApplet](#), this line also includes the words `implements Runnable`, so it looks like this:

```
public class OutputApplet extends Applet implements Runnable {  
    ...  
}
```

`Runnable` is an interface that makes it possible to run more than one thread at a time. A thread is a sequential flow of control, and it is possible for a program to be multithreaded, that is, to have many threads doing different things concurrently. The class `OutputApplet` implements the interface `Runnable` by defining the method `run`, the only method in `Runnable`. In our example the `run` method contains the JDBC code for opening a connection, executing a query, and getting the results from the result set. Since database connections can be slow, and can sometimes take several seconds, it is generally a good idea to structure an applet so that it can handle the database work in a separate thread.

Similar to a standalone application, which must have a `main` method, an applet must implement at least one `init`, `start`, or `paint` method. Our example applet defines a `start` method and a `paint` method. Every time `start` is invoked, it creates a new thread (named `worker`) to re-evaluate the database query. Every time `paint` is invoked, it displays either the query results or a string describing the current status of the applet.

As stated previously, the `run` method defined in `OutputApplet` contains the JDBC code. When the thread `worker` invokes the method `start`, the `run` method is called automatically, and it executes the JDBC code in the thread `worker`. The code in `run` is very similar to the code you have seen in our other sample code with three exceptions. First, it uses the class `Vector` to store the results of the query. Second, it does not print out the results but rather adds them to the `Vector` results for display later. Third, it likewise does not print out exceptions and instead records error messages for later display.

Applets have various ways of drawing, or displaying, their content. This applet, a very simple one that has only text, uses the method `drawString` (part of the `Graphics` class) to display its text. The method `drawString` takes three arguments: (1) the string to be displayed, (2) the `x` coordinate, indicating the horizontal starting point for displaying the string, and (3) the `y` coordinate, indicating the vertical starting point for displaying the string (which is below the text).

The method `paint` is what actually displays something on the screen, and in

OutputApplet.java , it is defined to contain calls to the method drawString . The main thing drawString displays is the contents of the Vector results (the stored query results). When there are no query results to display, drawString will display the current contents of the String message . This string will be "Initializing" to begin with. It gets set to "Connecting to database" when the method start is called, and the method setError sets it to an error message when an exception is caught. Thus, if the database connection takes much time, the person viewing this applet will see the message "Connecting to database" because that will be the contents of message at that time. (The method paint is called by AWT when it wants the applet to display its current state on the screen.)

The last two methods defined in the class OutputApplet, setError and setResults are private, which means that they can be used only by OutputApplet. These methods both invoke the method repaint , which clears the screen and calls paint . So if setResults calls repaint , the query results will be displayed, and if setError calls repaint , an error message will be displayed.

A final point to be made is that all the methods defined in OutputApplet except run are synchronized . The keyword synchronized indicates that while a method is accessing an object, other synchronized methods are blocked from accessing that object. The method run is not declared synchronized so that the applet can still paint itself on the screen while the database connection is in progress. If the database access methods were synchronized , they would prevent the applet from being repainted while they are executing, and that could result in delays with no accompanying status message.

To summarize, in an applet, it is good programming practice to do some things you would not need to do in a standalone application:

1. Put your JDBC code in a separate thread
2. Display status messages on the screen during any delays, such as when a database connection is taking a long time
3. Display error messages on the screen instead of printing them to System.out or System.err .

Running an Applet

Before running our sample applet, you need to compile the file OutputApplet.java . This creates the file OutputApplet.class , which is referenced by the file [OutputApplet.html](#).

The easiest way to run an applet is to use the appletviewer, which is included as part of the JDK. Simply follow the instructions below for your platform to compile and run OutputApplet.java :

UNIX

```
javac OutputApplet.java
appletviewer OutputApplet.html
```

Windows 95/NT

```
javac OutputApplet.java
appletviewer OutputApplet.html
```

Applets loaded over the network are subject to various security restrictions. Although this can seem bothersome at times, it is absolutely necessary for network security, and security is one of the major advantages of using the Java programming language. An applet cannot make network connections except to the host it came from unless the browser allows it. Whether one is able to treat locally installed applets as "trusted" also depends on the security restrictions imposed by the browser. An applet cannot ordinarily read or write files on the host that is executing it, and it cannot load libraries or define native methods.

Applets can usually make network connections to the host they came from, so they can work very well on intranets.

The JDBC-ODBC Bridge driver is a somewhat special case. It can be used quite successfully for intranet access, but it requires that ODBC, the bridge, the bridge native library, and JDBC be installed on every client. With this configuration, intranet access works from Java applications and from trusted applets. However, since the bridge requires special client configuration, it is not practical to run applets on the Internet with the JDBC-ODBC Bridge driver. Note that this is a limitation of the JDBC-ODBC Bridge, not of JDBC. With a pure Java JDBC driver, you do not need any special configuration to run applets on the Internet.

Lesson: New Features in the JDBC 2.0 API

The `java.sql` package that is included in the JDK 1.2 release (known as the JDBC 2.0 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 sections of this tutorial are written using the JDBC 1.0 API.

With the JDBC 2.0 API, you will be able 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 statements to the database as a unit, or batch
- Use the new SQL3 datatypes as column values

[Getting Set Up to Use the JDBC 2.0 API](#)

[Moving the Cursor in Scrollable Result Sets](#)

[Making Updates to Updatable Result Sets](#)

[Updating a Result Set Programmatically](#)

[Inserting and Deleting Rows Programmatically](#)

[Code Sample for Inserting a Row](#)

[Deleting a Row](#)

[Making Batch Updates](#)

[Using SQL3 Datatypes](#)

[Standard Extension Features](#)

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 let you move the cursor to a particular row and check the position of the cursor. Scrollable result sets make it possible to create a GUI (graphical user interface) tool for browsing result sets, which will probably be one of the main uses for this feature. Another use is moving to a row in order to update it.

Before you can take advantage of these features, however, you need to create a scrollable `ResultSet` object. The following line of code illustrates one way to create a scrollable `ResultSet` object:

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,  
                                     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 argument is one of three constants added to the `ResultSet` API to indicate the type of a `ResultSet` object: `TYPE_FORWARD_ONLY`, `TYPE_SCROLL_INSENSITIVE`, and `TYPE_SCROLL_SENSITIVE`. The second argument is one of two `ResultSet` constants for specifying whether a result set is read-only or updatable: `CONCUR_READ_ONLY` and `CONCUR_UPDATABLE`. The point to remember here is that if you specify a type, you must also specify whether it is read-only or updatable. Also, 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 of a `ResultSet` object, you will automatically get one that is `TYPE_FORWARD_ONLY` and `CONCUR_READ_ONLY` (as is the case when you are using only the JDBC 1.0 API).

You will get a scrollable `ResultSet` object if you specify one of the following `ResultSet` constants: `TYPE_SCROLL_INSENSITIVE` or `TYPE_SCROLL_SENSITIVE`. The difference between the two has to do with whether a result set reflects changes that are made to it while it is open and whether certain methods can be called to detect these 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. You might keep in mind, though, the fact that no matter what type of result set you specify, you are always limited by what your DBMS and driver actually provide.

Once you have a scrollable `ResultSet` object, `srs` in the previous example, you can use it to move the cursor around in the result set. Remember that when you created a new `ResultSet` object earlier in this tutorial, it had a cursor positioned before the first row. Even when a result set is scrollable, the cursor is 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 once, going from the first row to the last row, but now you have 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. You have already 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_SENSITIVE,
                                     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`. Then the method `previous` moves the cursor from the position after the last row to the last row, and then to the previous row with each iteration through the while loop. 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 will have the same values, but the rows are in the opposite order.

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 row indicated in their names. 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
```

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

```
srs.absolute(4);
int rowNum = srs.getRow(); // rowNum should be 4
srs.relative(-3);
int rowNum = srs.getRow(); // rowNum should be 1
srs.relative(2);
int 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 their names: `isFirst`, `isLast`, `isBeforeFirst`, `isAfterLast`. These methods all return a boolean and can therefore be used in a conditional statement. For example, the following code fragment tests to see whether the cursor is after the last row before invoking the method `previous` in a while loop. If the method `isAfterLast` returns false, the cursor is not after the last row, so the method `afterLast` is invoked. This guarantees that the cursor will be after the last row and that using the method `previous` in the while loop will cover every row in `srs`.

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

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.

Making Updates to Updatable Result Sets

Another new feature in the JDBC 2.0 API is the ability to update rows in a result set using methods in the Java programming language rather than having to send an SQL command. But before you can take advantage of this capability, you need to create a `ResultSet` object that is updatable. In order to do this, you supply the `ResultSet` constant `CONCUR_UPDATABLE` to the `createStatement` method, as you have seen in previous examples. The `Statement` object it creates 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. With a scrollable result set, you can move to rows you want to change, and if the type is `TYPE_SCROLL_SENSITIVE`, you can get the new value in a row after you have changed it.

```
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 an existing row from `uprs`, or modify a column value in `uprs`.

Updating a Result Set Programmatically

An update is the modification of a column value in the current row. Let's 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 shows another way to accomplish the update, this time using the JDBC 2.0 API:

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

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.

The `ResultSet`.`updateXXX` methods 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 datatype (`updateString`,

updateBigDecimal , updateInt , and so on) just as there are different getXXX methods for retrieving different datatypes.

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 and not just the result set, 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();
```

If you had moved the cursor to a different row before calling the method updateRow , the update would have been lost. If, on the other hand, you realized that the price should really have been 10.79 instead of 10.99, you could have cancelled the update to 10.99 by calling the method cancelRowUpdates . You have to invoke cancelRowUpdates before invoking the method updateRow ; once updateRow is called, calling the method cancelRowUpdates does nothing. Note that cancelRowUpdates cancels all of the updates in a row, so if there are many invocations of the updateXXX methods on the same row, you cannot cancel just one of them. The following code fragment first cancels updating the price to 10.99 and then updates it to 10.79:

```
uprs.last();
uprs.updateFloat("PRICE", 10.99);
uprs.cancelRowUpdates();
uprs.updateFloat("PRICE", 10.79);
uprs.updateRow();
```

In this example, only one column value was updated, but you can call an appropriate updateXXX method for any or all of the column values in a single row. The concept to remember is that updates and related operations apply to the row where the cursor is positioned. Even if there are many calls to updateXXX methods, it takes only one call to the method updateRow to update the database with all of the changes made in the current row.

If you want to update the price for COLOMBIAN_DECAF as well, you have to move the cursor to the row containing that coffee. Because the row for COLOMBIAN_DECAF immediately precedes the row for FRENCH_ROAST_DECAF , you can call the method previous to position the cursor on the row for COLOMBIAN_DECAF . The following code fragment changes the price in that row to 9.79 in both the result set and the underlying table in the database:

```
uprs.previous();
uprs.updateFloat("PRICE", 9.79);
uprs.updateRow();
```

All cursor movements refer to rows in a ResultSet object, not rows in the underlying database. If a query selects five rows from a database table, there will be five rows in the result set, with the first row being row 1, the second row being row 2, and so on. Row 1 can also be identified as the first, and, in a result set with five rows, row 5 is the last.

The ordering of the rows in the result set has nothing at all to do with the order of the rows in the base table. In fact, the order of the rows in a database table is indeterminate. The DBMS keeps track of which rows were selected, and it makes updates to the proper rows, but they may be located anywhere in the table. When a row is inserted, for example, there is no way to know where in the table it has been inserted.

Inserting and Deleting Rows Programmatically

In the previous section you saw how to modify a column value using methods in the JDBC 2.0 API rather than having to use SQL commands. With the JDBC 2.0 API, you can also insert a new row into a table or delete an existing row programmatically.

Let's suppose that our coffee house proprietor is getting a new variety from one of his coffee suppliers, The High Ground, and wants to add the new coffee to his database. Using the JDBC 1.0 API, he would write code that passes an SQL insert statement to the DBMS. The following code fragment, in which `stmt` is a `Statement` object, shows this approach:

```
stmt.executeUpdate("INSERT INTO COFFEES " +
                  "VALUES ('Kona', 150, 10.99, 0, 0)");
```

You can do the same thing without using any SQL commands by using `ResultSet` methods in the JDBC 2.0 API. Basically, after you have a `ResultSet` object with results from the table `COFFEES`, you can build the new row and then insert it into both the result set and the table `COFFEES` in one step. You build a new row in what is called the insert row, a special row associated with every `ResultSet` object. This row is not actually part of the result set; you can think of it as a separate buffer in which to compose a new row.

Your first step will be 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 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", 10.99);
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, 10.99);
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 but do not supply a value for every column in the row. If you fail to supply a value for a column that was defined to accept SQL NULL values, then the value assigned to that column is NULL . If a column does not accept null values, however, you will get an SQLException when you do not call an updateXXX method to set a value for it. This is also true if a 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 it is safer to select all columns. Especially if your table has hundreds or thousands of rows, you might want to use a WHERE clause to limit the number of rows returned by your SELECT statement.

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. You can, for instance, invoke any of the methods that put the cursor on a specific row, such as first , last , beforeFirst , afterLast , and absolute . You can also use the methods previous , relative , and moveToCurrentRow . Note that you can invoke moveToCurrentRow only when the cursor is on the insert row.

When you call the method moveToInsertRow , the result set records which row the cursor is sitting on, which is by definition the current row. As a consequence, the method

moveToCurrentRow can move 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.

Code Sample for Inserting a Row

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

Note: Tutorial reader Nedzad Hodzic reported that in this example, you cannot use `SELECT * FROM TAB_NAME`. You must name the columns; otherwise the `ResultSet` is not updateable. This is a bug he encountered when using an Oracle 8.1.6 database with Oracle driver OCI8.

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 *JDBC Database Access with Java*, but they are beyond the scope of this 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 same `Statement` object `stmt` and again using the query `SELECT * FROM COFFEES` .
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());
    }
}

```

Deleting a Row

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. All you do is 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();

```

The fourth row has been removed 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. For example, if you write an application meant to run with different databases, you should not write code that depends on there being a hole in a result set.

Seeing Changes in Result Sets

If you or anyone else modifies data in a `ResultSet` object, the change will always be visible if you close it and then reopen it. In other words, if you re-execute the same query, you will produce a new result set, based on the data currently in a table. This result set will naturally reflect changes anyone made earlier.

The question is whether you can see changes you or anyone else made while the `ResultSet` object is still open. (Generally, you will be most interested in the changes made by others.) The answer depends on the DBMS, the driver, and the type of `ResultSet` object you have.

With a `ResultSet` object that is `TYPE_SCROLL_SENSITIVE`, you can always see updates anyone makes to column values. You can usually see inserts and deletes, but the only way to be sure is to use `DatabaseMetaData` methods that return this information. (Refer to the second edition of *JDBC Database Access with Java* for information on getting metadata and for more details regarding 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(TRANSACTION_READ_COMMITTED);
```

With this isolation level, your `ResultSet` object 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 the higher the isolation level, the poorer the performance. And, as is always true of databases and drivers, you are limited to what they actually provide. Many programmers just use their database's default transaction isolation level. If you want more information about transaction isolation levels, you should consult a book on databases or the second edition of *JDBC Database Access with Java* .

In a `ResultSet` object that is `TYPE_SCROLL_INSENSITIVE` , you generally cannot see changes made to it while it is still open. Some programmers use only this type of `ResultSet` object because they want a consistent view of data and do not want to see

changes made by others.

You can use the method `refreshRow` to get the latest values for a row straight from the database. This method can be very 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 still available.)

```
Statement stmt = con.createStatement(
                                ResultSet.TYPE_SCROLL_SENSITIVE,
                                ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery(
    "SELECT COF_NAME, PRICE FROM COFFEES");
uprs.absolute(4);
Float price1 = uprs.getFloat("PRICE");
// do something. . .
uprs.absolute(4);
uprs.refreshRow();
Float price2 = uprs.getFloat("PRICE");
if (price2 > price1) {
    // do something. . .
}
```

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 multiple update statements to the database together as a unit can, in some situations, be much more efficient than sending each update statement separately. This ability to send updates as a unit, referred to as the batch update facility, is one of the features provided with the JDBC 2.0 API.

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 version of `executeUpdate`.

With the JDBC 2.0 API, Statement, PreparedStatement, and CallableStatement objects have the ability to maintain a list of commands that can be submitted together as a batch. They are created with an associated list, which is initially empty. You can add SQL commands to this list with the method `addBatch`, and you can empty the list with the method `clearBatch`. You send all of the commands in the list to the database with the method `executeBatch`. 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 batch updates. Remember that the table `COFFEES` has five columns: column `COF_NAME` of type `VARCHAR (32)`, column `SUP_ID` of type `INTEGER`, column `PRICE` of type `FLOAT`, column `SALES` of type `INTEGER`, and column `TOTAL` of type `INTEGER`. Each row he inserts will have values for the five columns in order. The code for inserting the new rows as a batch might look similar to 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();
```

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 sections [Disabling Auto-commit Mode](#) and [Committing a Transaction](#).) 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 a list of commands associated with it, and that list is empty.

```
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, which indicate how many rows were affected by each command, are stored in the array of `int`, `updateCounts`.

At this point `updateCounts` should contain four elements of type `int`. In this case, each `int` will be 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 empty this list of commands with the method `clearBatch`.

Batch Update Exceptions

There are two exceptions that can be thrown during a batch update operation: `SQLException` and `BatchUpdateException`.

All methods in the `JDBC` API will throw an `SQLException` object when there is a database access problem. In addition, the method `executeBatch` will throw an `SQLException` if you have used the method `addBatch` to add a command that returns a result set to the batch of commands being executed. Typically a query (a `SELECT` statement) will return a result set, but some methods, such as some of the `DatabaseMetaData` methods can also return a result set.

Just using the method `addBatch` to add a command that produces a result set does not cause an exception to be thrown. There is no problem while the command is just sitting in a `Statement` object's command list. But there will be a problem when the method `executeBatch` submits the batch to the DBMS to be executed. When each command is executed, it must return an update count that can be added to the array of update counts returned by the `executeBatch` method. Trying to put a result set in an array of update counts will cause an error and cause `executeBatch` to throw an `SQLException`. In other

words, only commands that return an update count (commands such as INSERT INTO , UPDATE , DELETE , CREATE TABLE , DROP TABLE , ALTER TABLE , and so on) can be executed as a batch with the executeBatch method.

If no SQLException was thrown, you know that there were no access problems and that all of the commands produce update counts. If one of the commands cannot be executed for some other reason, the method executeBatch will throw a BatchUpdateException . In addition to the information that all exceptions have, this exception contains an array of the update counts for the commands that executed successfully before the exception was thrown. Because the update counts are in the same order as the commands that produced them, you can tell how many commands were successful and which commands they are.

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 the SQLException information and the update counts contained in a BatchUpdateException object. Because getUpdateCounts returns an array of int , it uses a for loop to print each of the update counts.

```
try {
    // make some updates
} catch (BatchUpdateException b) {
    System.err.println("SQLException: " + b.getMessage());
    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] + " ");
    }
}
```

For the complete Batch Update program, see [BatchUpdate.java](#). The 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.

Using SQL3 Datatypes

The datatypes commonly referred to as SQL3 types are the new datatypes 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 datatypes into the Java programming language. With these new interfaces, you can work with SQL3 datatypes the same way you do other datatypes.

The new SQL3 datatypes give a relational database more flexibility in what can be used as a type 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 the SQL3 types. We will discuss them in more detail later.

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

Using SQL3 Datatypes

You retrieve, store, and update SQL3 datatypes the same way you do other datatypes. You use either `ResultSet. getXXX` or `CallableStatement. getXXX` methods to retrieve them, `PreparedStatement. setXXX` methods to store them, and `updateXXX` 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:

SQL3 type	getXXX method	setXXX method	updateXXX method
BLOB	getBlob	setBlob	updateBlob
CLOB	getClob	setClob	updateClob
ARRAY	getArray	setArray	updateArray
Structured type	getObject	setObject	updateObject
REF (structured type)	getRef	setRef	updateRef

For example, the following code fragment retrieves an SQL ARRAY value. For this example, 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 = 2238");
```

```
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 2238.

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",
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
pstmt.setClob(1, notes);
```

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.

Blob, Clob, and Array Objects

An important feature about `Blob`, `Clob`, and `Array` objects is that you can manipulate them without having to bring all of the data from the database server to your client machine. An instance of any of these types is actually a 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 improve performance dramatically.

You can use SQL commands and the JDBC 1.0 and 2.0 API with `Blob`, `Clob`, and `Array` objects just as if you were operating on the actual object in the database. If you want to work with any of them as an object in the Java programming language, however, you need to bring all their data over to the client, which we refer to as materializing the object. For example, if you want to use an SQL `ARRAY` object in an application as if it were an array in the Java programming language, you need to materialize the `ARRAY` object on the client and then convert it to an array in the Java programming language. Then you can use array methods in the Java programming language to operate on the elements of the array. The interfaces `Blob`, `Clob`, and `Array` all have methods for materializing the objects they represent. Refer to the second edition of *JDBC Database Access with Java* if you want more details or examples.

Struct and Distinct Types

SQL structured types and distinct types are the two datatypes 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.

An SQL structured type is similar to structured types in the Java programming language in that it has members, called attributes, that may be of any datatype. In fact, an attribute

may itself be another structured type. Here is an example of a simple definition creating a new SQL datatype:

```
CREATE TYPE PLANE_POINT
(
    X FLOAT,
    Y FLOAT
)
```

Unlike Blob , Clob , and Array objects, a Struct object contains values for each of the attributes in the SQL structured type and is not just a logical pointer to the object in the database. For example, suppose that a PLANE_POINT object is stored in column POINTS of table PRICES .

```
ResultSet rs = stmt.executeQuery(
    "SELECT POINTS FROM PRICES WHERE PRICE > 3000.00");
while (rs.next()) {
    Struct point = (Struct)rs.getObject("POINTS");
    // do something with point
}
```

If the PLANE_POINT object retrieved has an X value of 3 and a Y value of -5, the Struct object point will contain the values 3 and -5.

You might have noticed that Struct is the only type not to have a getXXX and setXXX method with its name as XXX. You must use getObject and setObject with Struct instances. This means that when you retrieve a value using the method getObject, you will get an Object in the Java programming language that you must explicitly cast to a Struct, as was done in the previous code example.

The second SQL type that a user can define in an SQL CREATE TYPE statement is a distinct type. An SQL distinct type is similar to a typedef in C or C++ in that it is a new type based on an existing type. Here is an example of creating a distinct type:

```
CREATE TYPE MONEY AS NUMERIC(10, 2)
```

This definition creates the new type called MONEY , which is a number of type NUMERIC that is always base 10 with two digits after the decimal point. MONEY is now a datatype in the schema in which it was defined, and you can store instances of MONEY in a table that has a column of type MONEY .

An SQL distinct type is mapped to the type in the Java programming language to which its underlying type would be mapped. For example, NUMERIC maps to java.math.BigDecimal , so the type MONEY maps to java.math.BigDecimal . To retrieve a MONEY object, you use ResultSet.getBigDecimal or CallableStatement.getBigDecimal ; to store a MONEY object, you use PreparedStatement.setBigDecimal .

SQL3 Advanced Features

Some aspects of working with SQL3 types can get quite complex. We mention some of the more advanced features here so that you will know about them, but a deeper

explanation is not appropriate for a basic tutorial. JDBC Database Access with Java contains a complete explanation of all JDBC features if you want to know more.

The interface Struct is the standard mapping for an SQL structured type. If you want to make working with an SQL structured type easier, you can map it to a class in the Java programming language. The structured type becomes a class, and its attributes become fields. You do not have to use a custom mapping, but it can often be more convenient.

Sometimes you may want to work with a logical pointer to an SQL structured type rather than with all the values contained in the structured type. This might be true, for instance, if the structured type has many attributes or if the attributes are themselves large. To reference a structured type, you can declare an SQL REF type that represents a particular structured type. An SQL REF object is mapped to a Ref object in the Java programming language, and you can operate on it as if you were operating on the structured type object that it represents.

Standard Extension Features

The package javax.sql, a standard extension to the Java programming language, provides these features:

Rowsets A rowset encapsulates a set of rows from a result set and may maintain an open database connection or be disconnected from the data source. A rowset is a JavaBeans™ component; it can be created at design time and used in conjunction with other JavaBeans components in a visual JavaBeans builder tool to construct an application. **JNDI™ for Naming Databases** The Java™ Naming and Directory Interface™ (JNDI) makes it possible to connect to a database using a logical name instead of having to hard code a particular database and driver. **Connection Pooling** A connection pool is a cache of open connections that can be used and reused, thus cutting down on the overhead of creating and destroying database connections. **Distributed Transaction Support** Support for distributed transactions allows a JDBC driver to support the standard two-phase commit protocol used by the Java Transaction API (JTA). This feature facilitates using JDBC functionality in Enterprise JavaBeans components.