



Trail: JDBC(TM) Database Access

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 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](#)



[Start of Tutorial](#) > [Start of Trail](#)

[Search](#)
[Feedback Form](#)

[Copyright](#) 1995-2003 Sun Microsystems, Inc. All rights reserved.



Trail: JDBC(TM) Database Access

Lesson: JDBC Basics

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.



[Start of Tutorial](#) > [Start of Trail](#) > [Start of Lesson](#)

[Search](#)
[Feedback Form](#)

[Copyright](#) 1995-2003 Sun Microsystems, Inc. All rights reserved.



Trail: JDBC(TM) Database Access

Lesson: JDBC Basics

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.



[Copyright](#) 1995-2003 Sun Microsystems, Inc. All rights reserved.

**Trail:** JDBC(TM) Database Access**Lesson:** JDBC Basics

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.



[Start of Tutorial](#) > [Start of Trail](#) > [Start of Lesson](#)

[Search](#)
[Feedback Form](#)

**Trail:** JDBC(TM) Database Access**Lesson:** JDBC Basics

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



[Start of Tutorial](#) > [Start of Trail](#) > [Start of Lesson](#)

[Search](#)
[Feedback Form](#)



Trail: JDBC(TM) Database Access

Lesson: JDBC Basics

Retrieving Values from Result Sets

We now show how you send the above `SELECT` statements from a program written in the Java programming language and how you get the results we showed.

JDBC returns results in a `ResultSet` object, so we need to declare an instance of the class `ResultSet` to hold our results. The following code demonstrates declaring the `ResultSet` object `rs` and assigning the results of our earlier query to it:

```
ResultSet rs = stmt.executeQuery(  
    "SELECT COF_NAME, PRICE FROM COFFEES");
```

Using the Method `next`

The variable `rs`, which is an instance of `ResultSet`, contains the rows of coffees and prices shown in the result set example above. In order to access the names and prices, we will go to each row and retrieve the values according to their types. The method `next` moves what is called a cursor to the next row and makes that row (called the current row) the one upon which we can operate. Since the cursor is initially positioned just above the first row of a `ResultSet` object, the first call to the method `next` moves the cursor to the first row and makes it the current row. Successive invocations of the method `next` move the cursor down one row at a time from top to bottom. Note that with the JDBC 2.0 API, covered in the next section, you can move the cursor backwards, to specific positions, and to positions relative to the current row in addition to moving the cursor forward.

Using the `getXXX` Methods

We use the `getXXX` method of the appropriate type to retrieve the value in each column. For example, the first column in each row of `rs` is `COF_NAME`, which stores a value of SQL type `VARCHAR`. The method for retrieving a value of SQL type `VARCHAR` is `getString`. The second column in each row stores a value of SQL type `FLOAT`, and the method for retrieving values of that type is `getFloat`. The following code accesses the values stored in the current row of `rs` and prints a line with the name followed by three spaces and the price. Each time the method `next` is invoked, the next row becomes the

current row, and the loop continues until there are no more rows in `rs` .

```
String query = "SELECT COF_NAME, PRICE FROM COFFEES";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
    String s = rs.getString("COF_NAME");
    float n = rs.getFloat("PRICE");
    System.out.println(s + "    " + n);
}
```

The output will look something like this:

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

Note that we use a curved arrow to identify output from JDBC code; it is not part of the output. The arrow is not used for results in a result set, so its use distinguishes between what is contained in a result set and what is printed as the output of an application.

Let's look a little more closely at how the `getXXX` methods work by examining the two `getXXX` statements in this code. First let's examine `getString` .

```
String s = rs.getString("COF_NAME");
```

The method `getString` is invoked on the `ResultSet` object `rs` , so `getString` will retrieve (get) the value stored in the column `COF_NAME` in the current row of `rs` . The value that `getString` retrieves has been converted from an SQL `VARCHAR` to a `String` in the Java programming language, and it is assigned to the `String` object `s` . Note that we used the variable `s` in the `println` expression shown above, that is, `println(s + " " + n)` .

The situation is similar with the method `getFloat` except that it retrieves the value stored in the column `PRICE` , which is an SQL `FLOAT` , and converts it to a Java `float` before assigning it to the variable `n` .

JDBC offers two ways to identify the column from which a `getXXX` method gets a value. One way is to give the column name, as was done in the example above. The second way is to give the column index (number of the column), with 1 signifying the first column, 2 , the second, and so on. Using the column number instead of the column name looks like

this:

```
String s = rs.getString(1);  
float n = rs.getFloat(2);
```

The first line of code gets the value in the first column of the current row of `rs` (column `COF_NAME`), converts it to a Java `String` object, and assigns it to `s`. The second line of code gets the value stored in the second column of the current row of `rs`, converts it to a Java `float`, and assigns it to `n`. Note that the column number refers to the column number in the result set, not in the original table.

In summary, JDBC allows you to use either the column name or the column number as the argument to a `getXXX` method. Using the column number is slightly more efficient, and there are some cases where the column number is required. In general, though, supplying the column name is essentially equivalent to supplying the column number.

JDBC allows a lot of latitude as far as which `getXXX` methods you can use to retrieve the different SQL types. For example, the method `getInt` can be used to retrieve any of the numeric or character types. The data it retrieves will be converted to an `int`; that is, if the SQL type is `VARCHAR`, JDBC will attempt to parse an integer out of the `VARCHAR`. The method `getInt` is recommended for retrieving only SQL `INTEGER` types, however, and it cannot be used for the SQL types `BINARY`, `VARBINARY`, `LONGVARBINARY`, `DATE`, `TIME`, or `TIMESTAMP`.

[Table 24, Methods for Retrieving SQL Types](#) shows which methods can legally be used to retrieve SQL types and, more important, which methods are recommended for retrieving the various SQL types. Note that this table uses the term "JDBC type" in place of "SQL type." Both terms refer to the generic SQL types defined in `java.sql.Types`, and they are interchangeable.

Using the Method `getString`

Although the method `getString` is recommended for retrieving the SQL types `CHAR` and `VARCHAR`, it is possible to retrieve any of the basic SQL types with it. (You cannot, however, retrieve the new SQL3 datatypes with it. We will discuss SQL3 types later in this tutorial.) Getting all values with `getString` can be very useful, but it also has its limitations. For instance, if it is used to retrieve a numeric type, `getString` will convert the numeric value to a Java `String` object, and the value will have to be converted back to a numeric type before it can be operated on as a number. In cases where the value will be treated as a string anyway, there is no drawback. Further, if you want an application to retrieve values of any standard SQL type other than SQL3 types, use the `getString` method.

Use of ResultSet.getXXX Methods to Retrieve JDBC Types

Note: If you have trouble reading this table, see [Use of ResultSet.getXXX: Table-Free Versions](#) for alternate views of the same information.

	T I N Y I N T	S M A L L I N T	I N T E G E R	B I G I N T	R E A L	F L O A T	D O U B L E	D E C I M A L	N U M E R I C	B I T	C H A R	V A R C H A R	L O N G V A R C H A R	B I N A R Y	V A R B I N A R Y	L O N G V A R B I N A R Y	D A T E	T I M E	T I M E S T A M P
getBytes	X	x	x	x	x	x	x	x	x	x	x	x	x						
getShort	x	X	x	x	x	x	x	x	x	x	x	x	x						
getInt	x	x	X	x	x	x	x	x	x	x	x	x	x						
getLong	x	x	x	X	x	x	x	x	x	x	x	x	x						
getFloat	x	x	x	x	X	x	x	x	x	x	x	x	x						
getDouble	x	x	x	x	x	X	X	x	x	x	x	x	x						
getBigDecimal	x	x	x	x	x	x	x	X	X	x	x	x	x						

getBoolean	x	x	x	x	x	x	x	x	x	X	x	x	x							
getString	x	x	x	x	x	x	x	x	x	x	X	X	x	x	x	x	x	x	x	x
getBytes														X	X	x				
getDate											x	x	x				X			x
getTime											x	x	x					X		x
getTimestamp											x	x	x				x	x		X
getAsciiStream											x	x	X	x	x	x				
getUnicodeStream											x	x	X	x	x	x				
getBinaryStream														x	x	X				
getObject	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

An "x" indicates that the `getXXX` method may legally be used to retrieve the given JDBC type.

An "X" indicates that the `getXXX` method is recommended for retrieving the given JDBC type.



[Start of Tutorial](#) > [Start of Trail](#) > [Start of Lesson](#)

[Search](#)
[Feedback Form](#)



Trail: JDBC(TM) Database Access

Lesson: JDBC Basics

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.



[Start of Tutorial](#) > [Start of Trail](#) > [Start of Lesson](#)

[Search](#)
[Feedback Form](#)



Trail: JDBC(TM) Database Access

Lesson: JDBC Basics

Use of ResultSet.getXXX: Table-Free Versions

Because the large table in [Retrieving Values from Result Sets](#)◆ can be hard to read (especially with screen readers), this page presents two alternate versions of the table's data. The first version displays the information you can get by looking down each column of the table. The second displays the information for each row of the table.

- [Version 1: How to read each JDBC type](#)
- [Version 2: Which types each ResultSet.getXXX method can read](#)

If you have any suggestions for improving this page, please [tell us](#).

Version 1: How to read each JDBC type

This section lists the ResultSet.getXXX methods recommended and allowed for retrieving data of each JDBC type.

TINYINT: `getBytes` (recommended)

Can also be read using `getShort`, `getInt`, `getLong`, `getFloat`, `getDouble`, `getBigDecimal`, `getBoolean`, `getString`, `getObject`

SMALLINT: `getShort` (recommended)

Can also be read using `getBytes`, `getInt`, `getLong`, `getFloat`, `getDouble`, `getBigDecimal`, `getBoolean`, `getString`, `getObject`

INTEGER: `getInt` (recommended)

Can also be read using `getBytes`, `getShort`, `getLong`, `getFloat`, `getDouble`, `getBigDecimal`, `getBoolean`, `getString`, `getObject`

BIGINT: `getLong` (recommended)

Can also be read using `getBytes`, `getShort`, `getInt`, `getFloat`, `getDouble`, `getBigDecimal`, `getBoolean`, `getString`, `getObject`

REAL: getFloat (recommended)

Can also be read using getByte, getShort, getInt, getLong, getDouble, getBigDecimal, getBoolean, getString, getObject

FLOAT: getDouble (recommended)

Can also be read using getByte, getShort, getInt, getLong, getFloat, getBigDecimal, getBoolean, getString, getObject

DOUBLE: getDouble (recommended)

Can also be read using getByte, getShort, getInt, getLong, getFloat, getBigDecimal, getBoolean, getString, getObject

DECIMAL: getBigDecimal (recommended)

Can also be read using getByte, getShort, getInt, getLong, getFloat, getDouble, getBoolean, getString, getObject

NUMERIC: getBigDecimal (recommended)

Can also be read using getByte, getShort, getInt, getLong, getFloat, getDouble, getBoolean, getString, getObject

BIT: getBoolean (recommended)

Can also be read using getByte, getShort, getInt, getLong, getFloat, getDouble, getBigDecimal, getString, getObject

CHAR: getString (recommended)

Can also be read using getByte, getShort, getInt, getLong, getFloat, getDouble, getBigDecimal, getBoolean, getDate, getTime, getTimestamp, getAsciiStream, getUnicodeStream, getObject

VARCHAR: getString (recommended)

Can also be read using getByte, getShort, getInt, getLong, getFloat, getDouble, getBigDecimal, getBoolean, getDate, getTime, getTimestamp, getAsciiStream, getUnicodeStream, getObject

LONGVARCHAR: getAsciiStream, getUnicodeStream (both recommended)

Can also be read using getByte, getShort, getInt, getLong, getFloat, getDouble, getBigDecimal, getBoolean, getString, getDate, getTime, getTimestamp, getObject

BINARY: getBytes (recommended)

Can also be read using getString, getAsciiStream, getUnicodeStream, getBinaryStream, getObject

VARBINARY: getBytes (recommended)

Can also be read using getString, getAsciiStream, getUnicodeStream, getBinaryStream, getObject

LONGVARBINARY: getBinaryStream (recommended)

Can also be read using getString, getBytes, getAsciiStream, getUnicodeStream, getObject

DATE: getDate (recommended)

Can also be read using getString, getTimestamp, getObject

TIME: getTime (recommended)

Can also be read using getString, getTimestamp, getObject

TIMESTAMP: getTimestamp (recommended)

Can also be read using getString, getDate, getTime, getObject

Version 2: Which types each ResultSet.getXXX method can read

This section lists the JDBC types that each ResultSet.getXXX method supports.

getBytes: TINYINT (recommended)

Can also read SMALLINT, INTEGER, BIGINT, REAL, FLOAT, DOUBLE, DECIMAL, NUMERIC, BIT, CHAR, VARCHAR, LONGVARCHAR

getShort: SMALLINT (recommended)

Can also read TINYINT, INTEGER, BIGINT, REAL, FLOAT, DOUBLE, DECIMAL, NUMERIC, BIT, CHAR, VARCHAR, LONGVARCHAR

getInt: INTEGER (recommended)

Can also read TINYINT, SMALLINT, BIGINT, REAL, FLOAT, DOUBLE, DECIMAL, NUMERIC, BIT, CHAR, VARCHAR, LONGVARCHAR

getLong: BIGINT (recommended)

Can also read TINYINT, SMALLINT, INTEGER, REAL, FLOAT, DOUBLE, DECIMAL, NUMERIC, BIT, CHAR, VARCHAR, LONGVARCHAR

getFloat: REAL (recommended)

Can also read TINYINT, SMALLINT, INTEGER, BIGINT, FLOAT, DOUBLE, DECIMAL, NUMERIC, BIT, CHAR, VARCHAR, LONGVARCHAR

getDouble: FLOAT, DOUBLE (both recommended)

Can also read TINYINT, SMALLINT, INTEGER, BIGINT, REAL, DECIMAL, NUMERIC, BIT, CHAR, VARCHAR, LONGVARCHAR

getBigDecimal: DECIMAL, NUMERIC (both recommended)

Can also read TINYINT, SMALLINT, INTEGER, BIGINT, REAL, FLOAT, DOUBLE, BIT, CHAR, VARCHAR, LONGVARCHAR

getBoolean: BIT (recommended)

Can also read TINYINT, SMALLINT, INTEGER, BIGINT, REAL, FLOAT, DOUBLE, DECIMAL, NUMERIC, CHAR, VARCHAR, LONGVARCHAR

getString: CHAR, VARCHAR (both recommended)

Can also read TINYINT, SMALLINT, INTEGER, BIGINT, REAL, FLOAT, DOUBLE, DECIMAL, NUMERIC, BIT, LONGVARCHAR, BINARY, VARBINARY, LONGVARBINARY, DATE, TIME, TIMESTAMP

getBytes: BINARY, VARBINARY (both recommended)

Can also read LONGVARBINARY

getDate: DATE (recommended)

Can also read CHAR, VARCHAR, LONGVARCHAR, TIMESTAMP

getTime: TIME (recommended)

Can also read CHAR, VARCHAR, LONGVARCHAR, TIMESTAMP

getTimestamp: TIMESTAMP (recommended)

Can also read CHAR, VARCHAR, LONGVARCHAR, DATE, TIME

getAsciiStream: LONGVARCHAR (recommended)

Can also read CHAR, VARCHAR, BINARY, VARBINARY, LONGVARBINARY

getUnicodeStream: LONGVARCHAR (recommended)

Can also read CHAR, VARCHAR, BINARY, VARBINARY, LONGVARBINARY

getBinaryStream: LONGVARBINARY (recommended)

Can also read BINARY, VARBINARY

getObject: (no recommended type)

Can read TINYINT, SMALLINT, INTEGER, BIGINT, REAL, FLOAT,
DOUBLE, DECIMAL, NUMERIC, BIT, CHAR, VARCHAR, LONGVARCHAR,
BINARY, VARBINARY, LONGVARBINARY, DATE, TIME, TIMESTAMP



[Start of Tutorial](#) > [Start of Trail](#) > [Start of Lesson](#)

[Search](#)
[Feedback Form](#)

[Copyright](#) 1995-2003 Sun Microsystems, Inc. All rights reserved.



Trail: JDBC(TM) Database Access

Lesson: JDBC Basics

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.



[Copyright](#) 1995-2003 Sun Microsystems, Inc. All rights reserved.



Trail: JDBC(TM) Database Access

Lesson: JDBC Basics

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.



[Start of Tutorial](#) > [Start of Trail](#) > [Start of Lesson](#)

[Search](#)
[Feedback Form](#)

[Copyright](#) 1995-2003 Sun Microsystems, Inc. All rights reserved.



Trail: JDBC(TM) Database Access

Lesson: JDBC Basics

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

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
```



[Start of Tutorial](#) > [Start of Trail](#) > [Start of Lesson](#)

[Search](#)
[Feedback Form](#)

**Trail:** JDBC(TM) Database Access**Lesson:** JDBC Basics

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.



[Start of Tutorial](#) > [Start of Trail](#) > [Start of Lesson](#)

[Search](#)
[Feedback Form](#)

[Copyright](#) 1995-2003 Sun Microsystems, Inc. All rights reserved.



Trail: JDBC(TM) Database Access

Lesson: JDBC Basics

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.



[Copyright](#) 1995-2003 Sun Microsystems, Inc. All rights reserved.

```
import java.sql.*;

public class TransactionPairs {

    public static void main(String args[]) {

        String url = "jdbc:mySubprotocol:myDataSource";
        Connection con = null;
        Statement stmt;
        PreparedStatement updateSales;
        PreparedStatement updateTotal;
        String updateString = "update COFFEES " +
                               "set SALES = ? where COF_NAME like ?";

        String updateStatement = "update COFFEES " +
                                  "set TOTAL = TOTAL + ? where COF_NAME like ?";
        String query = "select COF_NAME, SALES, TOTAL from COFFEES";

        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");

            updateSales = con.prepareStatement(updateString);
            updateTotal = con.prepareStatement(updateStatement);
            int [] salesForWeek = {175, 150, 60, 155, 90};
            String [] coffees = {"Colombian", "French_Roast",
                                "Espresso", "Colombian_Decaf",
                                "French_Roast_Decaf"};

            int len = coffees.length;
            con.setAutoCommit(false);
            for (int i = 0; i < len; i++) {
                updateSales.setInt(1, salesForWeek[i]);
                updateSales.setString(2, coffees[i]);
                updateSales.executeUpdate();

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

            con.setAutoCommit(true);

            updateSales.close();
            updateTotal.close();

            stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery(query);

            while (rs.next()) {
                String c = rs.getString("COF_NAME");
                int s = rs.getInt("SALES");
                int t = rs.getInt("TOTAL");
                System.out.println(c + "      " + s + "      " + t);
            }
        }
    }
}
```

```
    }  
  
    stmt.close();  
    con.close();  
  
} catch(SQLException ex) {  
    System.err.println("SQLException: " + ex.getMessage());  
    if (con != null) {  
        try {  
            System.err.print("Transaction is being ");  
            System.err.println("rolled back");  
            con.rollback();  
        } catch(SQLException excep) {  
            System.err.print("SQLException: ");  
            System.err.println(excep.getMessage());  
        }  
    }  
}  
  
}  
  
}
```



Trail: JDBC(TM) Database Access

Lesson: JDBC Basics

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.



[Start of Tutorial](#) > [Start of Trail](#) > [Start of Lesson](#)

[Search](#)
[Feedback Form](#)



Trail: JDBC(TM) Database Access

Lesson: JDBC Basics

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.



[Start of Tutorial](#) > [Start of Trail](#) > [Start of Lesson](#)

[Search](#)
[Feedback Form](#)

[Copyright](#) 1995-2003 Sun Microsystems, Inc. All rights reserved.



Trail: JDBC(TM) Database Access

Lesson: JDBC Basics

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
```



[Start of Tutorial](#) > [Start of Trail](#) > [Start of Lesson](#)

[Search](#)
[Feedback Form](#)

[Copyright](#) 1995-2003 Sun Microsystems, Inc. All rights reserved.

```
import java.sql.*;

public class CreateCoffees {
    public static void main(String args[]) {
        String url = "jdbc:mySubprotocol:myDataSource";
        Connection con;
        String createString;
        createString = "create table COFFEES " +
            "(COF_NAME VARCHAR(32), " +
            "SUP_ID INTEGER, " +
            "PRICE FLOAT, " +
            "SALES INTEGER, " +
            "TOTAL INTEGER)";

        Statement stmt;

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

        try {
            con = DriverManager.getConnection(url, "myLogin", "myPassword");
            stmt = con.createStatement();
            stmt.executeUpdate(createString);
            stmt.close();
            con.close();

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



Trail: JDBC(TM) Database Access

Lesson: JDBC Basics

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.



[Start of Tutorial](#) > [Start of Trail](#) > [Start of Lesson](#)

[Search](#)
[Feedback Form](#)

[Copyright](#) 1995-2003 Sun Microsystems, Inc. All rights reserved.

```
/**
 * This is a demonstration JDBC applet.
 * It displays some simple standard output from the Coffee database.
 */

import java.applet.Applet;
import java.awt.Graphics;
import java.util.Vector;
import java.sql.*;

public class OutputApplet extends Applet implements Runnable {
    private Thread worker;
    private Vector queryResults;
    private String message = "Initializing";

    public synchronized void start() {
        // Every time "start" is called we create a worker thread to
        // re-evaluate the database query.
        if (worker == null) {
            message = "Connecting to database";
            worker = new Thread(this);
            worker.start();
        }
    }

    /**
     * The "run" method is called from the worker thread. Notice that
     * because this method is doing potentially slow databases accesses
     * we avoid making it a synchronized method.
     */

    public void run() {
        String url = "jdbc:mySubprotocol:myDataSource";
        String query = "select COF_NAME, PRICE from COFFEES";

        try {
            Class.forName("myDriver.ClassName");
        } catch (Exception ex) {
            setError("Can't find Database driver class: " + ex);
            return;
        }

        try {
            Vector results = new Vector();
            Connection con = DriverManager.getConnection(url,
                "myLogin", "myPassword");

            Statement stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery(query);
            while (rs.next()) {
                String s = rs.getString("COF_NAME");
                float f = rs.getFloat("PRICE");
                String text = s + "      " + f;
                results.addElement(text);
            }

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

            setResults(results);
        } catch (SQLException ex) {
            setError("SQLException: " + ex);
        }
    }
}
```

```
    }  
}  
  
/**  
 * The "paint" method is called by AWT when it wants us to  
 * display our current state on the screen.  
 */  
  
public synchronized void paint(Graphics g) {  
    // If there are no results available, display the current message.  
    if (queryResults == null) {  
        g.drawString(message, 5, 50);  
        return;  
    }  
  
    // Display the results.  
    g.drawString("Prices of coffee per pound: ", 5, 10);  
    int y = 30;  
    java.util.Enumeration enum = queryResults.elements();  
    while (enum.hasMoreElements()) {  
        String text = (String)enum.nextElement();  
        g.drawString(text, 5, y);  
        y = y + 15;  
    }  
}  
  
/**  
 * This private method is used to record an error message for  
 * later display.  
 */  
  
private synchronized void setError(String mess) {  
    queryResults = null;  
    message = mess;  
    worker = null;  
    // And ask AWT to repaint this applet.  
    repaint();  
}  
  
/**  
 * This private method is used to record the results of a query, for  
 * later display.  
 */  
  
private synchronized void setResults(Vector results) {  
    queryResults = results;  
    worker = null;  
    // And ask AWT to repaint this applet.  
    repaint();  
}  
}
```

Output from query select NAME, PRICE from COFFEES