

You are here: [Home](#) > [Developers](#) > [View Source Articles](#) > Java View Source Article

Developer Program
[Membership](#)
[One-to-One Support](#)
[Newsgroups](#)

Developer Publications
[View Source](#)
[Developer News](#)

Documentation
[Technical Manuals](#)
[White Papers](#)
[TechNotes](#)
[Sample Code](#)
[FAQs](#)
[Books](#)

Technologies
[CORBA](#)
[Directory & LDAP](#)
[Dynamic HTML](#)
[Java](#)
[JavaScript](#)
[Linux](#)
[Security](#)
[SSJS](#)
[XML](#)

Developer Downloads
[Tools & SDKs](#)
[Patches](#)

iPlanet Products
[Technical Resources](#)

Adding Database Support With JDBC

A Primer for Java Programmers

By [Duane K. Fields](#)

[Send comments and questions](#) about this article to [View Source](#).

In my earlier article [The Nuts and Bolts of Relational Databases: A Primer for Web Developers](#), I explained the basics of relational databases. In this article, I'll describe how to use JDBC to add database support to your Java applications, applets, or servlets. If you've never developed database code in Java before, this article will help you get started.

JDBC is the Java API for communicating with databases. While Sun claims that JDBC isn't an acronym (seems you can't trademark an acronym!), the rest of the world believes that JDBC stands for Java Database Connectivity. JDBC specifies the interface necessary to connect to a database, execute SQL commands and queries, and interpret the results. This interface is both database-independent and platform-independent.

Don't confuse JDBC with the ODBC (Open Database Connectivity) API. Even though they have the same basic mission - providing vendor-transparent access to databases through a standard interface - the interface for ODBC is written in C rather than Java. The good news is that you can access data through a database's ODBC interface using JDBC, if necessary.

The JDBC classes are part of the built-in

```
java.sql package, which must be imported into any Java class from which you wish to access JDBC. Additional, optional extensions added with JDBC 2.0 can be found in the javax.sql package, if it's installed on your system.
```

DATABASE ACCESS

The JDBC API by itself can't talk to a database directly. JDBC only defines a database-independent interface and a collection of helper classes for handling results, errors, and the like. The actual database access is provided by a JDBC driver, a database-specific class that implements the JDBC interface defined by the API. This driver is provided by the database vendor or a third-party provider, usually at an additional cost.

JDBC drivers are available for most popular databases. If you don't have a JDBC driver for your system, you may be able to access your database through Sun's ODBC driver, a free, platform-independent driver that can communicate with any ODBC-compliant database. However, Sun is quick to point out that you get what you pay for, recommending that their ODBC driver not be used for anything particularly important to you.

JDBC drivers operate pretty much behind the scenes, and you don't typically have to worry about how they work. This approach allows the developers of JDBC drivers to implement all sorts of functionality in a truly application-independent way. For example, a JDBC driver might optimize your database connections, defer to an application server, or manage transactions

across multiple data sources. Your code doesn't have to change, as it deals only with the API. You can even develop on one database and deploy on another. Ah, the joys of object-oriented programming and abstraction!

JDBC 2.0 was recently released by Sun and includes a number of enhancements. A good number of these, however, were made at the interface level, meaning that they'll only become available once your database vendor has released a JDBC 2.0-compliant driver. Except where noted, all the material presented in this article refers to both the 1.0 and 2.0 releases of the API.

Specifying the JDBC Driver

In your program, you have to specify the JDBC driver that you'll be using. You do this through a simple, if strange-looking, operation:

```
Class.forName(JDBC-driver-class-name)
```

If, for example, you were using Sun's ODBC JDBC driver (a JDBC driver for communicating with ODBC-compliant databases), you would use this code:

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

The

`Class.forName()` method makes the Java virtual machine (JVM) load the driver class into memory. You might expect that you need to instantiate the driver class and use the reference to the driver to access the database, but JDBC doesn't work that way. When the driver is loaded into the JVM's memory, it registers itself with JDBC's `java.sql.DriverManager` class. (You can also create an anonymous instance of the driver and register it yourself, with the same effect.) You then use static methods of `DriverManager` to obtain a reference to a driver-specific `Connection` object, which in turn provides access to the database.

You can eliminate database-specific items in your code by specifying the database driver class at run time as a string value in a properties file or via a lookup service. If you're using more than one brand of database in your application, you'll need to register all of the necessary drivers with the `DriverManager`. The driver used to create the actual

`Connection` class will be determined by how you establish the connection to the database, as explained next.

Establishing a Connection

Once you've referenced the necessary driver classes, you're ready to establish a database connection, which will provide you with an instance of the

`Connection` class. JDBC follows the Java paradigm of using a URL to connect to a data resource. The exact format of the URL depends on the driver (which means that the `DriverManager` can tell from the URL format which driver will be used to manage the connection). In general, though, the URL follows formats like these:

```
jdbc:driver-id:database-id  
jdbc:driver-id://host/database-id
```

The URL for the JDBC connection is passed to one of the `DriverManager`'s

`getConnection()` methods. These static methods accept several different sets of arguments, depending on whether you need a user name and password to connect to the database. Here's how you would connect to an Oracle database, for example:

```
Class.forName("oracle.jdbc.driver.OracleDriver");  
String url = "jdbc:oracle:oci8@mcpsdb";  
String username = "dillinger";  
String password = "master";  
Connection conn = DriverManager.getConnection(url, username, password)
```

At least, that's the theory. In reality, there's a lot happening here that could generate exceptions, so the Java compiler will require you to take this into

account by using the proper

try/catch blocks. Thus, establishing the connection to the above-mentioned database would actually look more like the following:

```
Connection connection;
try {
    Class.forName("oracle.jdbc.driver.OracleDriver");
    String url = "jdbc:oracle:oci8@thehost";
    String username = "dillinger";
    String password = "master";
    connection = DriverManager.getConnection(url, username, password);
}
catch (ClassNotFoundException e) {
    System.err.println("Could not load database driver!");
}
catch (SQLException e) {
    System.err.println("Could not connect to the database!");
}
finally {
    try { if (connection != null) connection.close(); }
    catch (SQLException e) { }
}
```

Note that you use a

finally block - which in turn requires its own try/catch block - to ensure that the database connection is closed. Doing lots of nested error handling is just a fact of life in dealing with JDBC, but believe me, your code is better for it in the end. Once you have a valid Connection object, you can issue queries and commands to the database and read through the results.

Your ability to connect to any particular database will depend on the security configurations specified by your database administrator. Each database provides a number of configuration options to control which machines your database will accept connections from. If you're unable to establish a connection, be sure that the machine running your application has connection privileges with the database.

Simplifying Database Access With JNDI and DataSource

In ColdFusion and other template/scripting systems, you access a database through a single identifier that corresponds to a database connection (or connection pool) preconfigured by the system's administrator. This enables you to eliminate database connection information from your code, referring to your database sources by a logical name like

EmployeeDB or SalesDatabase. The details of connecting to the database aren't exposed to your code. Thus, only the resource description needs to be reconfigured if a new driver class becomes available, the database server moves, or the login information changes; any components or code referencing this named resource don't have to be touched.

JDBC 2.0 introduced the DataSource interface and the [Java Naming and Directory Interface](#) (JNDI) technology for naming and location services. If you're not familiar with JNDI, you might want to look into it. JNDI can be used to shield your application code from database details such as driver class, user name, password, and connection URI. To create a database connection with JNDI, you specify a resource name, which corresponds to an entry in a database or naming service, and receive back the information necessary to establish a connection with your database. This shields your code and supporting components from changes to the database's configuration.

Here's an example of creating a connection from a data source defined in the JNDI registry:

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("jdbc/SalesDB");
Connection con = ds.getConnection("username", "password");
```

We can improve upon this abstraction, and further simplify database access, through custom tags that use JNDI to allow simple access to named database resources in a manner similar to ColdFusion and other tag-style languages.

RETRIEVING DATA FROM A TABLE

My previous article covered the basics of SQL, including the fact that a

`SELECT` command returns a table of result values based on the query you give it. Before you can issue queries to the database through JDBC, you need to create a `Statement` object. While a `Statement` object can be reused across multiple SQL requests, the object itself is created by the `Connection` object and is therefore dependent on the database driver you're using. It's pretty straightforward, as you can see:

```
Statement statement = connection.createStatement();
```

Issuing a Query Through JDBC

Issuing a query through JDBC is easy once you have a

`Statement` object; you simply pass the query to the object's `executeQuery()` method. The table of results is returned from the `executeQuery()` method as an instance of JDBC's `ResultSet` object. Here's what a query of the products table used as an example in my previous article would look like:

```
String query = "SELECT ITEM_ID, DESCRIPTION FROM PRODUCTS_TABLE";  
ResultSet results = statement.executeQuery(query);
```

Again, in the real world this would need to be encased in a

`try/catch` block to catch any SQL exceptions that are thrown, but for the sake of clarity I'll leave out the error catching for now. The results of this query are stored in the `results` variable, which contains the information shown in Figure 1.

Figure 1. The contents of our result set

ITEM_ID	DESCRIPTION
100	Yellow grid bug
200	Blue grid bug
300	Red grid bug
400	Light cycle battery
500	Energy pod

The

`ResultSet` object exposes the contents of this table one row at a time and provides methods to access data from each column in the current row, as indicated by an internal cursor. When the result set is first created, the row cursor isn't at the first row, but rather just before it. Calling the `next()` method of `ResultSet` advances the cursor one row, returning `true` if the call was successful. For example, to advance our result set through its entire collection of rows we would do the following:

```
while (results.next()) {  
    System.out.println("Found another row!");  
}
```

When the cursor is positioned at a row you'd like to examine, you can call any of the dozens of data access methods supported by the

`ResultSet` object to retrieve data from the columns and get information about the results. These methods mostly follow the naming pattern `getDatatype()` - for example, `getString()`, `getInteger()`, or `getDouble()`. Each accepts an argument that specifies either the name or the index number of the column you want to read from. The row you read from is determined by where the cursor is currently positioned. For example, we can print the contents of our example table by continually calling the `next()` method until it returns `false`, indicating that we've fallen off the end of the result set:

```
int item_id;
String description;
while (results.next()) {
    item_id = results.getInteger("ITEM_ID");
    description = results.getString("DESCRIPTION");
    System.out.println("Item #" + item_id + " is: " + description);
}
```

We could also have referred to the columns in our products table by index number rather than by name. This can be useful in building applications that aren't dependent on any particular schema. In fact, another object that you can create is a

`ResultSetMetaData` object, which contains information about the results themselves. Using the `ResultSetMetaData` object, you can determine the amount and type of data held in the rows and columns of the result set, making it possible to handle completely arbitrary queries and database schemas.

As an aside, JFC aficionados may wonder why the JDBC API chooses to return

`ResultSet` objects rather than [TableModel objects](#), which are supposed to be the universal means of expressing tabular data in Java. The short answer is chronology: JDBC is a much older API than Swing and JFC, which defines `TableModel`. It's still handy to bring these two data models together - for example, by creating a `TableModel` object that takes a `ResultSet` object in its constructor.

Your result set maintains an active connection to the database behind the scenes - fetching new rows as needed (usually several at a time) to keep the cursor up to date. Not downloading all the data at once keeps large result sets from bogging your application down with network delay and reduces the amount of memory required.

Inserting, Updating, and Deleting Data

Of course, databases aren't read-only. Your Java code will need to be able to add new data to its table and alter the data that's already there. Like retrieving data, these tasks are accomplished with SQL commands. You issue these commands through the

`Statement` object as before, but through its `executeInsert()`, `executeUpdate()`, and `executeDelete()` methods. These methods don't return `ResultSet` objects; instead, they return an integer value that specifies the number of rows altered by the command.

Using Prepared Statements

Prepared statements enable you to develop an SQL query template that you can reuse to handle similar requests with different values. First you create the query, which can be any sort of SQL statement, leaving any variable values undefined. You then specify values for the undefined elements before executing the query, and repeat as necessary.

Prepared statements are defined from the

`Connection` object, just like regular `Statement` objects. In your SQL query, replace any variable values with a question mark. For example:

```
String query = "SELECT * FROM GAME_RECORDS WHERE SCORE > ? AND TEAM = "
PreparedStatement statement = connection.prepareStatement(query);
```

Before you can execute the statement, you must specify a value for each missing parameter. The

`PreparedStatement` object supports a number of different methods, each tied to setting a value of a specific type - `int`, `long`, `string`, and so on. Each method takes two arguments: an index value indicating which missing parameter you're specifying, and the value itself. The first parameter has an index value of 1 (not 0). To specify and execute a query that selects all scores higher than 10,000 for the Gold team, for example, you would use the following statements:

```
statement.setInt(1, 10000); // Specify the score value.
statement.setString(2, "Gold"); // Specify the team value.
ResultSet results = statement.execute();
```

Once you've defined a prepared statement, you can reuse it simply by changing parameters as needed; there's no need to create a new instance as long as the basic query is unchanged. Thus, you can execute several queries without having to create a new

`Statement` object. You can even share a single prepared statement among all of an application's components or a servlet's users. Not only is this more efficient in terms of object creation and memory allocation, but the resulting code is also cleaner and more easily understood.

There's yet another way to use prepared statements. Instead of hard-coding the value argument, you can import the value from a Bean,

`userBean`, that's been initialized from an input form. Staying with the same example:

```
statement.setInt(1, userBean.getScore()); // Specify the score value
statement.setString(2, userBean.getTeam()); // Specify the team value.
ResultSet results = statement.execute();
```

The alternative is to build up each SQL statement from strings, which can quickly get confusing, especially with complex queries. Consider our example again, this time without the benefit of a prepared statement:

```
Statement statement = connection.createStatement();
String query = "SELECT * FROM GAME_RECORDS WHERE SCORE > " +
    userBean.getScore() + " AND TEAM = '" + user.getTeam() +
    userBean.getTeam() + "'";
ResultSet results = statement.executeQuery(query);
```

You can see another, perhaps even more important, benefit of using prepared statements here. When you insert a value into a prepared statement with one of its setter methods, you don't have to worry about proper quoting of strings, escaping of special characters, and conversions of dates and other values into the proper format for your particular database. This is especially a boon for servlets that are likely to be collecting search terms and input directly from users through form elements and are particularly vulnerable to unpredictable input and special characters. Also, since each database might have its own peculiarities with regard to formatting, especially for dates, using prepared statements can help further distance your code from dealing with any one particular database.

Cleaning Up

When you've finished with a

`ResultSet` or `Connection` object, you should call its `close()` method to disconnect from the database and release that resource. Calling `close()` on a `Connection` object will implicitly close any result sets that are still open before disconnecting from the database.

DATABASE POWER

There's certainly lots more to know about relational databases and JDBC. The book [JDBC API Tutorial and Reference, Second Edition](#) is a hefty 1059 pages. Still, with what I've covered in this article you should be able to store and

retrieve data in simple database tables in your Java programs, search against that data, and even manage basic table relationships. I hope this article and my previous one will encourage you to explore the power databases can bring to your programs.

FURTHER RESOURCES

- [The Nuts and Bolts of Relational Databases: A Primer for Web Developers](#), *View Source*
- Seth White et al., [JDBC API Tutorial and Reference, Second Edition: Universal Data Access for the Java 2 Platform](#)
- George Reese, [Database Programming with JDBC and Java](#)

View Source wants your feedback!
[Write to us](#) and let us know
what you think of this article.

[Duane K. Fields](#) is a senior engineer for the E-Business Enablement group of IBM's Tivoli Systems, where he creates web-based applications with Java and JSP. He lives in Austin, Texas. His first book, [Web Development with JavaServer Pages](#), which he coauthored with Mark Kolb, will be available in May 2000.

(2.00)

Any sample code included above is provided for your use on an "AS IS" basis, under the [Netscape License Agreement - Terms of Use](#)