JDBC Short Course

by Magelang Institute

[Table of Contents]

The goal of this course is to provide an introduction to the JDBC interface for connecting Java programs with SQL-based databases. The course is interwoven with flexible exercises that, together with the course text, allow programmers to tailor the learning process to their particular learning style. This course covers the following main aspects of JDBC:

- How to connect to a database using the JDBC/ODBC bridge
- Sending queries to the database and getting results
- Examining metadata information from the database
- Error handling.
- Concurrency and efficiency issues

Short Course Prerequisites

Although there is a short primer on SQL, this course assumes that you are familiar with relational databases and SQL. In particular, the student should know how to define a table in the database they will be using for the exercises.

These course notes, applets, and exercises were developed and tested with Netscape Navigator 3.01 under Windows 95/NT 4.0. Other browsers and earlier versions of Navigator may have trouble running all the applets.

Short Course Format and Duration

This course consists of cross-linked course notes and flexible exercises that will take about eight hours to complete. Programmers that have experience with concurrent programming may finish sooner, while those for which these concepts are entirely new will want to proceed more slowly.

How to Take this Short Course

Because people tend to learn in different ways and have different backgrounds, this course is designed with many paths through the material. You can begin with the exercises and refer back to the notes, or you can begin with the Course Notes and follow the embedded links to appropriate exercises. It is possible to cover the course content in a depth-first, or breadth-first manner. For example, you may want to learn everything about a particular subject before moving on, or you may want to get a broad overview before exploring each subject in depth.

If you are not familiar with the integrated exercise concept, please read the associated help.

Entry Points:

- Course Notes
- Exercises

JDBC Short Course

Table of Contents

by Magelang Institute

Overview

Course Notes

Java Database Programming

- Introduction to JDBC
- A Complete Example
  - Creating a Database
    - exercises
  - Getting Information from a Database
    - exercise
  - Obtaining Result MetaData Type Information
    - exercises
- Connecting a Java program to a database
  - exercises
- Talking to a Database
  - Database Updates
  - Database Queries
    - exercises
  - Prepared Statements
    - exercise
- Metadata
  - Information about a database
    - exercises
  - Information about a table within a database
    - exercise
- Transactions
- Stored Procedures

- Java-SQL Type Equivalence
- JDBC Exception Types
  - SQLExceptions
  - SQLWarnings
  - Data Truncation
  - exercise
- SQL Conformance
  - exercise
- Enterprise APIs
- Resources

SQL Primer

- Creating Tables
- Accessing Columns
- Storing Information
- Resources

Training Index

JDBC Short Course

Java Database Programming

by Magelang Institute

[Table of Contents]

In this course, you will learn:

- What Java Database Connectivity (JDBC™) and Open Database Connectivity (ODBC) are
- How to use JDBC to connect to a relational database
- How to execute SQL queries on the database
- How to handle the results of a query
- How to view the structure of a database using metadata

Introduction to JDBC

SQL is a language used to create, manipulate, examine, and manage relational databases. Because SQL is an application-specific language, a single statement can be very expressive and can initiate high-level actions, such as sorting and merging data. SQL was standardized in 1992 so that a program could communicate with most database systems without having to change the SQL commands. Unfortunately, you must connect to a database before sending SQL commands, and each database vendor has a different interface, as well as different extensions of SQL. Enter ODBC.

ODBC, a C-based interface to SQL-based database engines, provides a consistent interface for communicating with a database and for accessing database metadata (information about the database system vendor, how the data is stored, and so on). Individual vendors provide specific drivers or "bridges" to their particular database management system. Consequently, thanks to ODBC and SQL, you can connect to a database and manipulate it in a standard way. It is no surprise that, although ODBC began as a PC standard, it has become nearly an industry standard.

Though SQL is well suited for manipulating databases, it is unsuitable as a general application language and programmers use it primarily as a means of communicating with databases--another language is needed to feed SQL statements to a database and process results for visual display or report generation. Unfortunately, you cannot easily write a program that will run on multiple platforms even though the database connectivity standardization issue has been largely resolved. For example, if you wrote a database client in C++, you would have to totally rewrite the client for each platform; that is to say, your PC version would not run on a Macintosh. There are two reasons for this. First, C++ as a language is not portable for the simple reason that C++ is not completely specified, for example, how many bits does an `int` hold? Second and more importantly, support libraries such as network access and GUI libraries are different on each platform. Enter Java.

You can run a Java program on any Java-enabled platform without even recompiling that program. The Java language is completely specified and, by definition, a Java-enabled platform must support a known core of libraries. One such library is JDBC, which you can think of as a Java version of ODBC, and is itself a growing standard. Database vendors are already busy creating bridges from the JDBC API to their particular systems. JavaSoft has also provided a bridge driver that translates JDBC to ODBC, allowing you to communicate with legacy databases that have no idea that Java exists. Using Java in conjunction with JDBC provides a truly portable solution to writing database applications.

The JDBC-ODBC bridge driver is just one of four types of drivers available to support JDBC connectivity. It comes packaged with the JDK 1.1 (and eventually with 1.1 browsers), or as a separate package for use with 1.0 systems.

A Complete Example

Running through a simple, but complete, example will help you grasp the overall concepts of JDBC. The fundamental issues encountered when writing any database application are:

- Creating a database. You can either create the database outside of Java, via tools supplied by the database vendor, or via SQL statements fed to the database from a Java program.
- Connecting to an ODBC data source. An ODBC data source is a database that is registered with the ODBC driver. In Java you can use either the JDBC to ODBC bridge, or JDBC and a vendor-specific bridge to connect to the datasource.
- Inserting information into a database. Again, you can either enter data outside of Java, using database-specific tools, or with SQL statements sent by a Java program.
- Selectively retrieving information. You use SQL commands from Java to get results and then use Java to display or manipulate that data.

Creating a Database

For this example, consider the scenario of tracking coffee usage at the MageLang University Cafe. A weekly report must be generated for University management that includes total coffee sales and the maximum coffee consumed by a programmer in one day. Here is the data:

Coffee Consumption at ***Cafe Jolt***, MageLang University
"Caffeinating the World, one programmer at a time"

Programmer	Day	# Cups
Gilbert	Mon	1
Wally	Mon	2
Edgar	Tue	8
Wally	Tue	2
Eugene	Tue	3
Josephine	Wed	2

Eugene	Thu	3
Gilbert	Thu	1
Clarence	Fri	9
Edgar	Fri	3
Josephine	Fri	4

To create this database, you can feed SQL statements to an ODBC data source via the JDBC-ODBC bridge. First, you will have to create an ODBC data source. You have many choices—you could, for example, connect an Oracle or Sybase database. For simplicity and to cover the largest single audience, create a text file as an ODBC datasource to use for this example. Call this ODBC data source CafeJolt.

To enter the data into the CafeJolt database, create a Java application that follows these steps:

1. Load the JDBC-ODBC bridge. You must load a driver that tells the JDBC classes how to talk to a data source. In this case, you will need the class JdbcOdbcDriver:

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

   This can also be specified from the command line via the jdbc.drivers system property:

   ```
   java -Djdbc.drivers=sun.jdbc.odbc.JdbcOdbcDriver AProgram
   ```
2. Connect to a data source. A URL is used to connect to a particular JDBC data source. See Section 3.1.2 URLs in General USE and 3.1.3 JDBC URLs in the JDBC Guide for more information. Given that you have loaded the JDBC-ODBC bridge driver, URLs should be in the following form jdbc:odbc:*data-source-name*. Using the DriverManager class, you request a connection to a URL and the DriverManager selects the appropriate driver; here, only the driver JdbcOdbcDriver is loaded.

   ```
   Connection con = DriverManager.getConnection(
           URL,
           username,
           password);
   ```

   Where the *username* and *password* are empty strings, "", in this case because text files acting as ODBC data sources cannot have such attributes.
3. Send SQL statements to create the table. Ask the connection object for a Statement object:

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

   Then, execute the following SQL statement to create a table called JoltData.

   ```
   create table JoltData (
           programmer varchar (32),
           day char (3),
           cups integer,
           variety varchar (20));
   ```

The Java code to do this is:

```
stmt.execute(   "create table JoltData ("+
                "programmer varchar (32),"+
                "day char (3),"+
                "cups integer);"
            );
```

After you have created the table, you can the insert the appropriate values such as:

```
insert into JoltData values ('Gilbert', 'Mon', 1);
insert into JoltData values ('Wally', 'Mon', 2);
insert into JoltData values ('Edgar', 'Tue', 8);
...
```

Here is the Java source for a complete application to create table JoltData and insert the required rows.

Review what you have done so far. After creating a data source visible to ODBC, you connected to that source via the JDBC-ODBC bridge and sent a series of SQL statements to create a table called JoltData filled with rows of data. You can examine the contents of your "database" file by looking at file JoltData with a text editor. It will look like this:

```
"programmer","day","cups"
"Gilbert","Mon",1
"Wally","Mon",2
"Edgar","Tue",8
"Wally","Tue",2
"Eugene","Tue",3
"Josephine","Wed",2
"Eugene","Thu",3
"Gilbert","Thu",1
"Clarence","Fri",9
"Edgar","Fri",3
"Josephine","Fri",4
```

The ODBC-text driver will also create a file called schema.ini containing metadata:

```
[JoltData]
ColNameHeader=True
CharacterSet=OEM
Format=CSVDelimited
Col1=programmer Char Width 32
Col2=day Char Width 3
Col3=cups Integer
```

exercises

1. Getting Started.
2. Using JDBCTest.
3. Connecting to an ODBC datasource without JDBCTest.

Getting Information from a Database

To retrieve information from a database, use SQL select statements via the Java `Statement.executeQuery` method, which returns results as rows of data in a `ResultSet` object. The results are examined row-by-row using the `ResultSet.next` and `ResultSet.get`*XXX* methods.

Consider how you would obtain the maximum number of cups of coffee consumed by a programmer in one day. In terms of SQL, one way to get the maximum value is to sort the table by the `cups` column in descending order. The `programmer` column is selected, so the name attached to the most coffee consumption can also be printed. Use the SQL statement:

```
SELECT programmer, cups FROM JoltData ORDER BY cups DESC;
```

From Java, execute the statement with:

```
ResultSet result = stmt.executeQuery(
  "SELECT programmer,
  cups FROM JoltData ORDER BY cups DESC;");
```

The `cups` column of the first row of the result set will contain the largest number of cups:

Clarence	9
Edgar	8
Josephine	4
Eugene	3
Eugene	3
Edgar	3
Wally	2
Wally	2
Josephine	2
Gilbert	1
Gilbert	1

Examine the `ResultSet` by:

1. "Moving" to the first row of data. Perform:

   ```
   result.next();
   ```
2. Extracting data from the columns of that row. Perform:

   ```
   String name = result.getString("programmer");
   int cups = result.getInt("cups");
   ```

The information can be printed easily via:

```
System.out.println("Programmer "+name+
        " consumed the most coffee: "+cups+" cups.");
```

resulting in the following output:

```
Programmer Clarence consumed the most coffee: 9 cups.
```

Computing the total sales for the week is a matter of adding up the `cups` column. Use an SQL `select` statement to retrieve the `cups` column:

```
result = stmt.executeQuery(
```

```
        "SELECT cups FROM JoltData;");
```

Peruse the results by calling method next until it returns false, indicating that there are no more rows of data:

```
// for each row of data
cups = 0;
while(result.next()) {
        cups += result.getInt("cups");
}
```

Print the total number of cups sold:

```
System.out.println("Total sales of
  "+cups+" cups of coffee.");
```

The output should be:

```
Total sales of 38 cups of coffee.
```

Here is the Java source for a complete application to examine the JoltData table and generate the report.

exercise

1. Selecting.

Obtaining Result MetaData Type Information

You will occasionally need to obtain type information about the result of a query. For example, the SQL statement:

```
SELECT * from JoltData
```

will return a ResultSet with the same number of columns (and rows) as the table, JoltData. If you do not know how many columns there are beforehand, you must use metadata via the ResultSetMetaData class to find out. Continuing the *Cafe Jolt* scenario, determine the number and type of columns returned by the same SQL query

```
SELECT programmer, cups FROM JoltData ORDER BY cups DESC;
```

First, perform the usual execute method call:

```
ResultSet result = stmt.executeQuery(
    "SELECT programmer,
    cups FROM JoltData ORDER BY cups DESC;");
```

Then obtain the column and type metadata from the ResultSet:

```
ResultSetMetaData meta = result.getMetaData();
```

You can query the ResultSetMetaData easily to determine how many columns there are:

```
int columns = meta.getColumnCount();
```

and then walk the list of columns printing out their name and type:

```
int numbers = 0;
```

```
for (int i=1;i<=columns;i++) {
  System.out.println (meta.getColumnLabel(i) + "\t"
                    + meta.getColumnTypeName(i));
  if (meta.isSigned(i)) { // is it a signed number?
        numbers++;
  }
}
System.out.println ("Columns: " +
  columns + " Numeric: " + numbers);
```

Here is the Java source for a complete application to print out some metadata associated with the results of the query.

exercises

1. Connecting to an ODBC datasource without JDBCTest.
2. Using MetaData.

Connecting a Java program to a database

Currently, there are two choices for connecting your Java program to a data source. First, you can obtain a JDBC driver from your database vendor that acts as a bridge from JDBC to their database connection interface. Second, JavaSoft provides a JDBC-ODBC bridge called class JdbcOdbcDriver and, hence, you can connect to any ODBC data source.

Once you have established a JDBC database link, open a connection to that data source through a Connection object obtained via DriverManager.getConnection, which selects the appropriate driver for talking with that source. All ODBC data sources are identified via a URL in the form:

jdbc:odbc:*data-source-name*

The getConnection method returns a Connection to the specified source using the JDBC-ODBC driver. For example, to connect to an ODBC source called mage with a user name of parrt and a password of mojava, you would use:

```
// load the JDBC-ODBC bridge by referencing it
try {
  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
} catch (Exception e) {
  System.out.println(
    "Failed to load JDBC/ODBC driver.");
  return;
}
// get a connection
try {
  con = DriverManager.getConnection(
        "jdbc:odbc:mage",
        "parrt",
        "mojava");
} catch (Exception e) {
  System.err.println("problems connecting to "+URL);
}
```

Given a connection, you can create statements, execute queries, and so on.

See the Getting Started exercise for specific information about setting up ODBC data sources on a PC.

Exercises

1. Getting Started.
2. Using JDBCTest.
3. Connecting to an ODBC datasource without JDBCTest.

Talking to a Database

Given a connection to a database, you can send SQL statements to manipulate that database. Using the Connection.createStatement method, obtain a Statement object and then execute method executeQuery or executeUpdate. JDBC does not put any restrictions on the SQL you send via the execute methods. but you must ensure that the data source you are connecting to supports whatever SQL you are using. To be JDBC-compliant, however, the data source must support at least SQL-2 Entry Level capabilities.

Database Updates

Assuming the variable con contains a valid Connection object obtained from the method DriverManager.getConnection, simple SQL update statements (SQL INSERT, UPDATE or DELETE) can be sent to your database by creating a Statement and then calling method executeUpdate. For example, to create a table called Data with one row of data, use the following:

```
Statement stmt = con.createStatement();
// create table with name and height columns
stmt.executeUpdate(
  "create table Data (
    name varchar (32), height integer);");
stmt.executeUpdate(
  "insert into Data values ('John', 68);");
con.close();
// close connection, committing transaction.
```

The method executeUpdate returns the number of rows affected with 0 for SQL statements that return nothing.

Database Queries

In order to query a database (via the SQL SELECT statement), use the method executeQuery, which returns a ResultSet object. The ResultSet object returned is never null and contains the rows of data selected by the SQL statement. For example, the following code fragment selects two columns of data from our table called Data in ascending height order:

```
ResultSet result = stmt.executeQuery(
  "SELECT name, height
    FROM Data ORDER BY height ASC;");
```

The rows of resulting data are accessed in order, but the elements in the various columns can be accessed in any order. However, for maximum portability among databases, JavaSoft recommends that the columns be accessed in order from left-to-right, and that each row be read only once. There is a "row cursor" in the result that points at the current row. The method ResultSet.next moves the cursor from row to row. Before reading the first row, call method next to initialize the cursor to the first row. The following code fragment shows how to read the first two rows of data and print them out.

```
String name;
int height;
if ( result.next() ) {  // move to first row
  name = result.getString("name");
  height = result.getInt("height");
  System.out.println(name+":"+height);
}
if ( result.next() ) {  // get second row
  name = result.getString("name");
  height = result.getInt("height");
  System.out.println(name+":"+height);
}
```

The method next returns false when another row is not available.

Note that column names are *not* case-sensitive, and if more than one column has the same name, the first one is accessed. Where possible, the column index should be used. You can ask the ResultSet for the index of a particular column if you do not know it beforehand.

```
int nameCol = result.findColumn ("name");
int heightCol = result.findColumn ("height");
```

Information about the properties of a ResultSet column is provided by the class ResultSetMetaData and returned by the ResultSet.getMetaData method. See the section on MetaData for details.

exercises

1. Selecting.
2. Using MetaData.

Prepared Statements

When performing the same operation multiple times, use a PreparedStatement for runtime efficiency, which is *precompiled* by the SQL engine (assuming your data source supports this feature). Prepared statements are also useful when you have lots of arguments to specify for a particular SQL command.

PreparedStatement is an extension of Statement and, consequently, behaves like a Statement except that you create them with the method Connection.prepareStatement, instead of the method Connection.createStatement:

```
PreparedStatement prep = con.prepareStatement(
    "INSERT into Data values (?, ?)");
```

The IN arguments, indicated by '?', can be filled by in by set*XXX* methods. Ensure that the parameter (indicated by an index number starting from 1) you are setting matches the type of the value you are passing. For example, for a table called Data with two columns, name of type varchar and height of type integer, the parameters to prep can be set via:

```
prep.setString(1, "Jim");
prep.setInt(2, 70);
```

Finally, execute the the prepared statement with the parameters set most recently via the executeUpdate method:

```
if (prep.executeUpdate () != 1) {
    throw new Exception ("Bad Update");
}
```

exercise

1. Command-Line Guestbook.

Metadata

You can access information about the database as a whole, or about a particular query ResultSet. This section describes how DatabaseMetaData and ResultSetMetaData objects are obtained and queried.

Information about a database

When you need to know about the capabilities, or the vendor, of a database, ask the associated Connection object for its metadata:

```
Connection con = DriverManager.getConnection(
    "jdbc:odbc:mydatasource",
    "user", "password");
DatabaseMetaData md = con.getMetaData();
```

There are many questions you can ask. For example, the following code fragment asks the database for its product name and how many simultaneous connections can be made to it.

```
if (md==null) {
  System.out.println("No Database Meta Data");
} else {
  System.out.println("Database Product Name   : " +
        md.getDatabaseProductName());
  System.out.println("Allowable active connections: "+
        md.getMaxConnections());
}
```

See the DatabaseMetaData API for more information.

Exercises

1. Connecting to an ODBC datasource without JDBCTest.
2. Connecting with Properties.
3. SQL Warning/Exception Handling.

Information about a table within a database

To find out the number and types of the columns in a table accessed via a ResultSet, obtain a ResultSetMetaData object.

```
ResultSet result = ...;
ResultSetMetaData meta = result.getMetaData();
int numbers = 0;
int columns = meta.getColumnCount();
System.out.println ("Columns: " + columns);
```

exercise

1. Using MetaData.

Transactions

A transaction is a set of statements that have been executed and committed or rolled back. To commit a transaction, call the method commit on the appropriate connection; use the rollback to remove all changes since the last commit. By default, all new connections are in auto-commit mode, which means that each "execute" is a complete transaction. Call Connection.setAutoCommit to change the default. Any locks held by a transaction are released upon the method commit.

```
Connection con = DriverManager.getConnection(...);
con.setAutoCommit(false);
Statement s = con.createStatement();
s.executeUpdate("SQL statement 1");
s.executeUpdate("SQL statement 2");
s.executeUpdate("SQL statement 3");
con.commit();
//transaction (3 statements) committed here
```

All JDBC-compliant drivers support transactions. Check the DatabaseMetaData associated with the appropriate connection to determine the level of transaction-support a database provides.

Stored Procedures

A stored procedure is a block of SQL code stored in the database and executed on the server. The CallableStatement interface allows you to interact with them. Working with CallableStatement objects is very similar to working with PreparedStatements. The procedures have parameters and can return either ResultSets or an update count. Their parameters can be either input or output parameters. Input parameters are set via the set*XXX* methods. Output parameters need to be registered via the CallableStatement.registerOutParamter method. Stored procedures need to be supported by the database in order to use them. You can ask the DatabaseMetaData if it supports it via the method supportsStoredProcedures.

To demonstrate this interaction, return to Cafe Jolt. And, instead of a weekly total, the manager asks for the daily total of a particular day of the week. You can create a stored procedure to help, but this is usually created by the database developer, not the applications programmer. Once the procedure is created, the user does not need to know how it works internally, just how to call it.

Like any other SQL statement, you need a Connection and a Statement to create the procedure:

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

Then execute the SQL statement:

```
CREATE PROCEDURE getDailyTotal
     @day char(3), @dayTotal int output
     AS
     BEGIN
        SELECT @dayTotal = sum (cups)
        FROM JoltData
        WHERE day = @day
     END
```

The Java code is:

```
stmt.execute ("CREATE PROCEDURE getDailyTotal " +
     "@day char(3), @dayTotal int output " +
     "AS " +
     "BEGIN " +
     "   SELECT @dayTotal = sum (cups) " +
     "   FROM JoltData " +
     "   WHERE day = @day " +
     "END"
);
```

Once created, you call it through a CallableStatement:

```
CallableStatement cstmt = con.prepareCall (
  "{call getDailyTotal (?, ?)}");
cstmt.setString (1, "Mon");
cstmt.registerOutParameter (2, java.sql.Types.INTEGER);
cstmt.executeUpdate();
System.out.println ("Total is " + cstmt.getInt (2));
```

The exact syntax to create a stored procedure may differ between database vendors. Also, if there are no output parameters for the stored procedure, you can it by using a PreparedStatement. And, if there happens to be no input or output parameters, you can use a Statement.

Java-SQL Type Equivalence

For each get*XXX* method, the JDBC driver must convert between the database data type and a Java equivalent--the driver will not let you perform an invalid data conversion (for example, String to integer), but will let you get everything as a String. The common conversions are as follows:

Common Java - SQL Type Equivalence

Java method	SQL Type
getInt	INTEGER
getLong	BIG INT
getFloat	REAL
getDouble	FLOAT
getBignum	DECIMAL
getBoolean	BIT
getString	VARCHAR
getString	CHAR
getDate	DATE
getTime	TIME
getTimestamp	TIME STAMP

getObject	*any type*

For example, given a ResultSet containing rows of names and dates, you can use getString and getDate to extract the information:

```
ResultSet result = stmt.executeQuery(
  "SELECT name, whatDay FROM ...");
result.next();
String name1 = result.getString("name");
Date whatDay1 = result.getDate("whatDay");
```

JDBC Exception Types

JDBC provides three types of exceptions:

- SQLException
- SQLWarning
- DataTruncation

SQLExceptions

The SQLException is the basis of all JDBC exceptions. It consists of three pieces of information, a String message, like all children of Exception, another String containing the XOPEN SQL state (as described by specification), and a driver/source specific int for an additional error code.

In addition, multiple SQLException instances can be chained together.

SQLWarnings

The SQLWarning class is similar to SQLException, however it is considered a noncritical error and is not thrown. It is up to the programmer to poll for SQLWarning messages through the getWarnings methods of Connection, ResultSet, and Statement. If you do not poll for them, you will never receive them. Also, the next time something is done with a Connection, ResultSet, or Statement, the previous warnings are cleared out.

Data Truncation

The DataTruncation class is a special type of SQLWarning. It is reported with other SQLWarning instances and indicates when information is lost during a read or write operation. To detect a truncation, it is necessary to perform an instanceof DataTruncation check on each SQLWarning from a getWarnings chain.

Exercise

1. SQL Warning/Exception Handling.

SQL Conformance

Although SQL is a standard, not all JDBC drivers support the full ANSI92 grammar. Luckily, you can determine the level of conformance by asking. The DatabaseMetaData object contains three methods that report the grammar level supported by a driver.

- supportsANSI92EntryLevelSQL
  All JDBC-compliant drivers must return true.
- supportsANSI92IntermediateSQL

- supportsANSI92FullSQL

In addition to multiple ANSI92 support levels for JDBC drivers, there are multiple SQL support levels for ODBC sources. You can determine the level of these too by asking. The DatabaseMetaData object contains three methods that report the grammar level supported by a database.

- supportsMinimumSQLGrammar
- supportsCoreSQLGrammar
- supportsExtendedSQLGrammar

These levels determine which specific SQL operations, or which options of those operations, you can perform.

Also, when using the JDBC-ODBC bridge, the ODBC driver can further restrict your capabilities; for instance, you cannot use prepared statements with the Text File ODBC Driver.

Based upon the levels available, it may be beneficial to provide alternative execution paths for certain operations. For instance, when populating a table with values from another table, the core SQL grammar permits a SELECT clause as part of the INSERT clause. This would be much quicker than performing a select loop with an insert for each record, which is necessary if you are relying on the minimum SQL grammar.

Exercise

1. SQL Warning/Exception Handling.

Enterprise APIs

JDBC is just one part of what JavaSoft calls the *Java Enterprise APIs*. The remaining three parts are Remote Method Invocation (RMI) for Java-to-Java communications, via RPC-like calls, Java IDL for Java-to-CORBA connectivity through OMG's Interface Definition Language specification, and Java Naming and Directory Interface (JNDI) for directory services support. Closely related is Java Object Serialization, which permits programs to send objects across streams for persistence.

- RMI
- Serialization
- Java IDL
- JNDI

Resources

Some web-based resources:

- JDBC Guide
- JDBC Home

and books:

- *Java Database Programming with JDBC* by Prantik Patel and Karl Moss (Coriolis Group ISBN 1-57610-056-1)
- *Database Programming with JDBC and Java* by George Reese (O'Reilly ISBN 1-56592-270-0)

[This page was updated: 13-Dec-99]

Products & APIs
Developer Connection
Docs & Training
Online Support
Community Discussion
Industry News
Solutions Marketplace
Case Studies

Training Index

JDBC Short Course

SQL Primer

by Magelang Institute

[Table of Contents]

This section is meant to serve as a SQL refresher to help you along with the exercises. It is not meant to be a *tell-all* resource for SQL. It takes you through the basic commands necessary for CRUD operations.

- C - Create
- R - Read
- U - Update
- D - Delete

Creating Tables

Use the CREATE TABLE statement when you want to create a table. Because creating tables is such an important operation, it only requires minimum conformance. However, some datasources, such as Text ODBC sources, only support the simplest column elements, with little or no constraint support.

```
CREATE TABLE <table name>
  (<column element> [, <column element>]...)
```

A column element is of the form:

```
<column name> <data type>
   [DEFAULT <expression>]
   [<column constraint> [, <column constraint>]...]
```

A column constraint is of the form:

```
NOT NULL |
  UNIQUE |
  PRIMARY KEY
```

Example:

```
CREATE TABLE java (
   version_name varchar (30),
   major_version int,
   minor_version int,
   release_date date);
```

Use the DROP TABLE statement when you want to drop a table. Like CREATE TABLE, it only requires minimum conformance.

```
DROP TABLE <table name>
```

Accessing Columns

Use the SELECT statement when you want to retrieve a set of columns. The set may be from one or more tables, and you can specify the criteria to determine which rows to retrieve. Most of the available clauses are available with minimum conformance. Additional capabilities are available with the core grammar.

```
SELECT [ALL | DISTINCT] <select list>
   FROM <table reference list>
   WHERE <search condition list>
   [ORDER BY <column designator> [ASC | DESC]
          [, <column designator> [ASC | DESC]]...]
```

The select list usually contains a comma-separated list of columns or an '*' to select all of them.

```
SELECT version_name, release_date from java;
```

If your driver supports core compliance, you can also use the GROUP BY, HAVING, and UNION clauses.

Storing Information

Use the INSERT statement when you want to insert rows. It too can provide different capabilities depending upon the conformance level supported.

```
INSERT INTO <table name>
  [(<column name> [, <column name>]...)]
  VALUES (<expression> [, <expression>]...)
```

For example:

```
INSERT INTO java VALUES
  ('2.0Beta', 2, 0, 'Aug-1-1997');
```

If the core grammar is supported, you can use a SELECT clause to load multiple rows at a time.

Use the UPDATE statement when you want to update rows. It only requires the minimum grammar.

```
UPDATE <table name>
   SET <column name = {<expression> | NULL}
    [, <column name = {<expression> | NULL}]...
   WHERE <search condition>
```

Use the DELETE statement when you want to remove rows. It only requires the minimum grammar.

```
DELETE FROM <table name>
   WHERE <search condition>
```

Resources

Some web-based resources:

- Access FAQ
- dbANYWHERE FAQ
- Oracle FAQ
- Sybase FAQ
- Tina London's Guide to SQL

and books:

- *Teach Yourself SQL in 14 Days* by Bryan Morgan and Jeff Perkins (Sams Publishing ISBN 0-67230-855-X)
- *Understanding the New SQL: A Complete Guide* by Jim Melton and Alan Simon (Morgan Kaufman ISBN 1-55860-245-3)
- *LAN Times Guide to SQL* by James R. Groff & Paul N. Weinberg (McGraw-Hill ISBN 0-07882-026-X)