

LESSON 31

JDBC-EXAMPLES & EXERCISES

Introduction

SQL data types and Java data types are not exactly the same. See Table for a list-ing of the basic SQL data types and their equivalents in the Java programming language.

Objective

In this lesson we will populate a database and then start raising queries. Further we look at some examples using Metadata, and finally will look at scrollable and updatable Result Sets.

Lesson in detail

Let's look at the difference between SQL data types and their corresponding Java language types.

SQL data type

Java data type

INTEGER or INT

int

SMALLINT

short

NUMERIC (m,n),DECIMAL(m,n)orDEC(m,n)

java.sql.Numeric

FLOAT(n)

double

REAL

float

DOUBLE

double

CHARACTER(n)orCHAR(n)

String

VARCHAR(n)

String

BOOLEAN

boolean

DATE

java.sql.Date

TIME

java.sql.Time

TIMESTAMP

java.sql.Timestamp

BLOB

java.sql.Blob

CLOB

java.sql.Clob

ARRAY

java.sql.Array

Advanced SQL Types (JDBC 2)

In addition to numbers, strings, and dates, many databases can store *large objects* such as images or other data. In SQL, binary

large objects are called BLOBs, and character large objects are called CLOBs. The `getBlob` and `getClob` methods return objects of type `Blob` and `Clob`. These classes have methods to fetch the bytes or characters in the large objects.

A SQL ARRAY is a sequence of values. For example, in a Student table, you can have a Scores column that is an ARRAY OF INTEGER. The `getArray` method returns an object of type `java.sql.Array` (which is different from the `java.lang.reflect.Array` class that we discussed). The `java.sql.Array` interface has methods to fetch the array values.

NOTE: The BLOB, CLOB, and ARRAY types are features of SQL3. Java platform support for BLOB and CLOB has been greatly enhanced in JDBC2, and support for ARRAY is a new feature in JDBC 2.

When you get a blob or an array from a database, the actual contents are only fetched from the database when you request individual values. This is a useful performance enhancement, since the data can be quite voluminous.

Some databases are able to store user-defined structured types. JDBC 2 supports a mechanism for automatically mapping structured SQL types to Java objects. However, in this introductory chapter, we do not discuss blobs, arrays, and user-ed types any further.

java.sql.DriverManager

Static `getConnection(String url, String user, String password)` establishes a connection to the given database and returns a `Connection` object.

Parameters:	url	the URL for the database
	user	the database logon ID
	password	the database logon password

java.sql.Connection

`Statement createStatement()`

Creates a statement object that can be used to execute SQL queries and updates without parameters.

`void close ()`

immediately closes the current connection.

`void setAutoCommit(boolean b)`

sets the auto commit mode of this connection to `b`. If `autocommit` is true, all statements are committed as soon as their execution is completed.

`boolean getAutoCommit()`

gets the `autocommit` mode of this connection.

`void commit ()`

commits all statements that were issued since the last commit.

`void rollback ()`

undoes the effect of all statements that were issued since the last commit.

Java.sql.Statement

ResultSet executeQuery(String sql)

executes the SQL statement given in the string and returns a ResultSet to view the query result.

Parameters: sql the SQL query

int executeUpdate(String sql)

executes the SQL INSERT, UPDATE, or DELETE statement specified by the-string. Also used to execute DDL (Data Definition Language) statements such as CREATE TABLE. Returns the number of records affected.

Parameters: sql the SQL statement

.void cancel ()

creates a thread to cancel a JDBC statement that is being executed.

```
java.sql.ResultSet
```

boolean next()

makes the current row in the result set move forward by one. Returns false after the last row. Note that you must call this method to advance to the first row.

Xxx getXxx(int columnNumber)

Xxx getXxx(String columnName)

(Xxx is a type such as int, double, String, Date, etc.)

return the value of the column with column index column Number or with column names, converted to the specified type. Not all type conversions legal. See documentation for details.

int findColumn(String columnName)

gives the column index associated with a column name. void close ()

void close()

immediately closes the current result set.

String getSQLState()

gets the SQLState formatted, using the X/Open standard.

int getErrorCode()

gets the vendor-specific exception code.

SQLException getNextException()

gets the exception chained to this one. It may contain more information about the error.

Populating a Database

We now want to write our first, real, JDBC program. Of course, it would be nice if we could execute some of the fancy queries that we discussed earlier. Unfortunately, we have a problem: right now, there is no data in the database. And you won't find a database file on the CD-ROM that you can simply copy onto your hard disk for the database program to read, because no database file format lets interchange SQL relational databases from one vendor to another. SQL does not have anything to

do with files. It is a language to issue queries and updates to a database. How the database executes these statements most efficiently and what file formats it uses toward that goal are entirely up to the implementation of the database. Database vendors try very hard to come up with clever strategies for query optimization and data storage, and different vendors arrive at different mechanisms. Thus, while SQL statements are portable, the underlying data representation is not.

To get around our problem, we provide you with a small set of data in a series of text files. The first program reads such a text file and creates a table. The first line of the file contains the column names and types. The remaining lines of the input are the data, and we insert the lines into the table. Of course, we use SQL statements and JDBC to create the table and insert the data.

Specifically, the program reads data from a text file in a format such as Publisher_Id char(5) , Name char(30), URL char(80) '01262', 'Academic Press', 'www.apnet.com' '18835', 'Coriolis', 'www.coriolis.com/ '

First line of the text file lists the names and types of the columns. The following lines list the data to be inserted, in comma-delimited format. MakeDB program reads the first line and turns it into a CREATE TABLE statement such as

```
CREATE TABLE Publishers (Publisher_Id char(5), Name char(30), URL char(80))
All other input lines become INSERT statements such as INSERT INTO Publishers VALUES ('01262', 'Academic Press', 'www.apnet.com.')
```

At the end of this section, you can see the code for the program that reads a text file and populates a database table. Even if you are not interested in looking at the implementation, you must run this program if you want to execute the more interesting examples in the next two sections. Run the program as follows:

```
java MakeDB Books
```

```
java MakeDB Authors
```

```
java MakeDB Publishers
```

```
java MakeDB BooksAuthors
```

Before running the program, check the file MakeDB.properties. It looks like this

```
jdbc.drivers=com.pointbase.jdbc.jdbcDriver
```

```
jdbc.url=jdbc:pointbase:COREJAVA
```

```
jdbc.username=PUBLIC
```

```
jdbc.password=PUBLIC
```

These values work for the PointBase database; change them if you use another database.

Caution: Make sure that both the database driver and the current directory are on the class path. Alternatively, start up the program as `java -classpath paths MakeDB tableName`

The following steps provide an overview of the MakeDB program.

1. Connect to the database. The getConnection method reads the properties in the file MakeDB.properties and adds the jdbc.drivers property to the system properties. The driver manager uses the

jdbc.drivers property to load the appropriate database driver. The getConnection method uses the jdbc.url, jdbc.username, and jdbc.password properties to open the database connection.

- Obtain the file name from the table name by appending the extension dat (e.g., the data for the Books table is stored in the file Books.dat).
- Read in the column names and types and construct a CREATE TABLE command. Then execute that command:


```
String line = in.readLine();
String command = "CREATE TABLE " + tableName + "
(" + line + ")";
stmt.executeUpdate(command);
```

 Here, we use executeUpdate, not executeQuery, because this statement has no result.
- For each line in the input file, execute an INSERT statement:


```
command = "INSERT INTO " + tableName
+ " VALUES (" + line + ")";
stmt.executeUpdate(command);
```
- After all elements have been inserted, run a SELECT * FROM tableName query, using the showTable method to show the result. This method shows that the data has been successfully inserted. To find the number of columns in the result set, we need the getColumnCount method of the ResultSet:MetaData class. We discuss metadata in detail later in this chapter.

Next exercise for you is to :

Try to enter and run the following program and execute it .

Listing 31.1: MakeDB.java

```
import java.net.*;
import java.sql.*;
import java.io.*;
import java.util.*;

class MakeDB
{
    public static void main (String args[])
    { try
      {Connection con = getConnection();
        Statement stmt = con.createStatement();

        String tableName = " ";
        if (args.length > 0)
            tableName = args[0];
        else
            { System.out.println("Usage: MakeDB TableName");
              System.exit (0) ;
            }
    }
}
```

```
BufferedReader in = new BufferedReader(new
    FileReader(tableName + ".dat"));
createTable(tableName, in, stmt);
showTable(tableName, stmt);
in.close() ;
    stmt.close() ;
    con.close();
}
catch (SQLException ex)
{ System.out.println ("SQLException:");
  while (ex != null)
  { System.out.println ("_SQLState: "
    + ex.getSQLState());
    System.out.println ("Message:
    + ex.getMessage());
    System.out.println ("Vendor:
    + ex.getErrorCode());
    ex = ex.getNextException();
    System.out.println ("");
  }
}
catch (IOException ex)
{ System.out.println("Exception: + ex);
  ex.printStackTrace ();
}

public static Connection getConnection()
throws SQLException, IOException
{ Properties props = new Properties();
String fileName = "MakeDB.properties"; FileInputStream in =
new FileInputStream(fileName); props.load (in) ;
String drivers = props.getProperty("jdbc.drivers");
if (drivers != null)
System.setProperty(" jdbc.drivers", drivers);
String url = props.getProperty("jdbc.url");
String username = props.getProperty("jdbc.username");
String password = props.getProperty("jdbc.password");
return
    DriverManager. getConnection (url, username, password);
}

public static void createTable(String tableName,
BufferedReader in, Statement stmt)
throws SQLException, IOException
{ String line = in.readLine();
String command = "CREATE TABLE" + tableName
+ "(" + line + ")";

stmt.executeUpdate(command);
```

