

Database
Management
System

LINTER®

Version 5.9

JDBC

Relational Expert Systems



Table of Contents

Introduction3

Driver Characteristics4

Main Driver Functions 4

Data Type Compatibility 4

DB Access Scripts (Methods) 5

JDBC Driver Types..... 5

Input and Output Data7

java.sql.CallableStatements 7

java.sql.Connection 8

java.sql.Driver 9

java.sql.DriverManager 10

java.sql.PreparedStatement..... 10

java.sql.ResultSet..... 11

java.sql.Statement 13

java.sql.DatabaseMetaData 14

java.sql.ResultSetMetaData..... 14

Standard Classes 14

Driver Installation and Launch15

Driver Installation 15

Driver Launch 15

 Launching the Client Driver..... 15

 Launching the Server Driver 15

Introduction

Linter's JDBC driver is based on the Javasoft JDBC 1.01 standard, a part of JDK 1.1.5. The text provides information on JDBC standard compliance and the specifics of Linter's implementation of the standard for every JDBC driver interface. Additionally, installation and launch of the driver are described.

This reference is for the use of application programmers developing applications using network and Internet access to relational data bases. Java Data Base Connectivity (JDBC) is a standard interface used to access remote databases through the Internet using the Java language. Utilization of JDBC allows the application developer to create database- and platform-independent applets.

A Java-program can be developed as an applet that loads on the client via the Internet and is run when needed. Alternatively, it may be an application permanently located on the client. In both instances, the JDBC interface allows the Java-application to access remote databases, send queries, and receive results. It is necessary to remember that security constraints may limit operation of the applet. Therefore, the Web browser must be configured to permit network access to the applet. In practice, the JDBC interface is a collection of abstract classes (interfaces, in Java terminology) which must be defined for every data source. Therefore, a high-level abstract JDBC layer and an immediate low-level layer are possible. A high-level abstraction layer is used in application JDBC interfaces, which have access to numerous databases as well as a variety of query and data manipulation methods. Application interfaces allow for a higher abstraction layer, describing only the method, not the implementation, of declarations.

The immediate JDBC layer, unique for every database, is implemented via a specific JDBC driver.

Similar to ODBC, developers implement the JDBC interface using the driver manager (class `DriverManager`, the only one implemented by the `java.sql*` standard class package developers), which in turn supports numerous drivers allowing communication with various databases.

Driver Characteristics

Main Driver Functions

Linter's JDBC driver provides the following functions:

- 1) Transfer a query to the DB in the form of a string. This permits use of SQL structures specific to a DB and/or its JDBC-driver.
- 2) Execution of an SQL-query execution based on the X/Open and SQL Access Group (SAG) SQL CAE 1992 specifications.
- 3) Receive SQL-query processing results.
- 4) Return query processing termination codes.
- 5) Support for standard data types.
- 6) Form static and dynamic SQL-statements.
- 7) Send and receive data values in the format specified by the application.

Data Type Compatibility

The driver provides the following data types:

<u>DBMS Linter</u>	<u>Java equivalent</u>
Integer	Int (class Integer)
INTEGER	Int (class Integer)
Word	Unsigned short
WORD	Unsigned short
Longint	Long (class Long)
LONG	Long (class Long)
LONGINT	Long (class Long)
Byte	Byte
BYTE	Byte
Real	Float (class Float)
REAL	Float (class Float)
Longreal	Double (class Double)
LONGREAL	Double (class Double)
Boolean	Boolean
BOOLEAN	Boolean
CHAR	String
NUMERIC	BigDecimal
DECIMAL	BigDecimal

DB Access Scripts (Methods)

The procedure for accessing a DB (usually located on a remote server and not on a client machine) depends on whether the application is a network loaded applet or an independent applet resident on the client machine

For a network applet, the DB access script follows this sequence:

- 1) Loads the applet, as a byte-code, onto the client machine as part of the web-document.
- 2) Launches the applet on the client virtual machine.
- 3) The applet requests the JDBC driver manager that is located on the client machine.
- 4) The applet accesses the DB server using the TCP/IP protocol.

In case of an independent application the script is as follows:

- The client virtual machine launches the application.
- The application requests the JDBC driver that is located on the client machine.
- The application accesses the DB server using the TCP/IP protocol.

JDBC Driver Types

All JDBC drivers may be divided into the following classes:

- 1) The JDBC to ODBC bridge enables DB access using the ODBC drivers. For this Java driver, the ODBC driver and, in most cases, the DBMS client must be loaded on every machine on which the driver is to be used.
- 2) A driver in which the client side is only partially implemented in Java. These Java drivers usually use a dynamic link library (DLL) to translate the Java function calls into some other language. Drivers of this type translate Java calls into the client application interface of a specific DBMS. Like the bridge drivers, these drivers load some binary code onto every client machine.
- 3) A driver in which the client side is fully implemented in Java. The server side, if any, may be written in another language.
- 4) This is a driver written completely in Java. It converts the JDBC calls into a DBMS independent network protocol. The server then converts the network protocol into the DBMS protocol. This intermediate network application connects all Java clients with various databases. The specific DBMS protocol depends entirely on the developer. As a rule, this type of driver is the most flexible JDBC option. In drivers of this type, ODBC is sometimes used on the server to provide in-depth integration into the DBMS.
- 5) A driver in which both the client side and the server side, if any, are implemented in Java. All-Java drivers with a unique-to-the-driver protocol convert JDBC calls directly into the DBMS network protocol. This approach permits sending calls from the clients directly to the DBMS server without the intermediate conversion into an independent network protocol. This solution is practical in instances when the client configuration is strictly controlled, as in typical intranet applications. As most of these database protocols are non-standard, developers are responsible for providing drivers of this type.

The Linter JDBC driver described below belongs to the fourth class. All the standard interfaces are implemented in Java and do not require the presence of some binary code on every client

The server side of the driver is written in C for speed of execution It translates intermediate network protocol calls into DBMS calls and sends back replies.

Input and Output Data

Since Java is an object-oriented language, under application interface collection of classes and interfaces (in Java language terminology) is meant. These classes and interfaces are described in the `java.sql` package. The JDBC driver is a collection of classes, which implement the JDBC interfaces.

A complete description of the JDBC interface may be found in the JDBC 1.01. Programmers Reference: JavaSoft Standard. This document illustrates the specifics of the JDBC driver implementation for Linter.

java.sql.CallableStatements

Use

The `java.sql.CallableStatement` interface is used to execute stored procedures. It works with parametric SQL expressions.

Specifics

The argument values correspond to the JDBC 1.01 specification.

Example

```
import java.sql.*;
import jdbc.LinJdbc.*;

public class StoredProcDemo
{
    static String QueryD = new String("drop procedure fact1;");

    static String QueryC = "create procedure fact1()";

    static String QueryA = "alter procedure fact1(in x numeric default 1)
result numeric"+
        " for debug"+
        " declare"+
        " var r numeric;"+
        " exception badparam for custom 10;"+
        " code"+
        " if x < 0 then"+
        " signal badparam;"+
        " endif"+
        " if x <= 1 then"+
        " r :=1;"+
        " else"+
        " call fact(x-1) into r;"+
        " r := r*x;"+
        " endif"+
        " return r;"+
        " exceptions"+
        " when others then"+
        " resignal;"+
        " end;";

    public static void main (String[] argv)
    {

        Connection con;
        Statement st;
        try{

            Driver d =
Driver)Class.forName("jdbc.LinJdbc.LinterDriver").newInstance();
```

```

        System.out.println(" Driver found. Now connecting to
database... ");

        con =
DriverManager.getConnection("jdbc:Linter:195.98.69.49:1070:local",
"SYSTEM","MANAGER");
        st =con.createStatement();
        try{
        try{
            System.out.println("Create Procedure\n");
            System.out.println("Query=\n"+QueryC);
            st.executeUpdate(QueryC);
        }catch(SQLException ex)
        {
            do{
                System.out.print("Error occured: code =");
                System.out.println(ex.getErrorCode());
                System.out.print("          message= ");
                System.out.println(ex.getMessage());
                ex=ex.getNextException();
            }while(ex!=null);
        }

        System.out.println("Alter Procedure\n");
        System.out.println("Query=\n"+QueryA);
        st.executeUpdate(QueryA);

        System.out.println("Drop Procedure\n");
        System.out.println("Query=\n"+QueryD);
        st.executeUpdate(QueryD);

    }catch(SQLException ex)
    {
        //Обработка ошибок
        do{
            System.out.print("Error occured: code  =");
            System.out.println(ex.getErrorCode());
            System.out.print("          message= ");
            System.out.println(ex.getMessage());
            ex=ex.getNextException();
        }while(ex!=null);
        con.close();
    }
}catch(Exception ex)
{
    System.out.println("Error occured: "+ex.getMessage());
}
}
}

```

java.sql.Connection

Use

The `java.sql.Connection` interface determines the characteristics and the status of the connection to the DB. It also provides tools for transaction control and the level of isolation control. This interface is used to send queries to the DB and to receive query results. The class methods `commit`, `rollback`, and `setAutoCommit` control transactions. The `CallableStatement`, created by the `prepareCall` method of the `Connection` interface, controls stored procedures. The `PreparedStatement` objects (`prepareStatement` method) control pre-translated queries. The `Statement` objects control regular queries. The `Connection` interface may also be used to optimize the query execution process.

The `Connection` interface uses `getMetaData`, which returns a `DatabaseMetaData` object, method for extracting information about the DB. It provides information containing database table descriptions, supported SQL-grammar, stored procedures, connection possibilities, etc.

Specifics

The argument values correspond to the JDBC 1.01 specification

URL specification:

`jdbc:Linter:[remote url]:[remote port]:[local node]`

<code>remote url</code>	the IP-address of the remote node; if none entered, a local is used.
<code>remote port</code>	the port number (default 1070) of the application.
<code>local node</code>	a name from the <code>nodetab</code> table on the remote node or a key word if Linter and the application are running on separate servers.

Example

```
import java.sql.*;
import jdbc.LinJdbc.*;
public class Connect {
    public static void main (String[ ] args)
        { try {
            Driver d =(Driver)Class.forName("jdbc.LinJdbc.
            LinterDriver").newInstance( );
            String      address      = "jdbc:Linter:195.98.69.49:1070:local";
// 195.98.69.49      IP address of Linter JDBC server (linapid)
// 1070      socket number
// local      The word "local" must appear here or the name
//      of the remote Linter node from NODETAB
            String user      = "SYSTEM";
            String password= "MANAGER";
            System.out.println("Driver found. Now connecting to
            database ... ");
            Connection con =
            DriverManager.getConnection(address,user,password);
            /* ... */
            con.close( );
        }
        Catch (Exception e){
            System.out.println("Caught :"+e+" mess
            = "+e.getMessage( ) );
            e.printStackTrace( );
        }
    }
}
```

java.sql.Driver

Use

Extracts driver information, checks address, and connects to a database (creation of the Connection object).

Specifics

The argument values correspond to the JDBC 1.01 specification. To use the Linter JDBC driver, driver registration, using the following command, is necessary:

```
Class.forName("jdbc.LinJDBC.LinterDriver").newInstance( );
```

java.sql.DriverManager

Use

The `java.sql.DriverManager` interface loads drivers and creates new connections to a database. This is the main JDBC interface. It determines the correct selection and initialization of a driver in given conditions and for a given DBMS.

The `java.sql.DriverManager` interface includes both the driver and the driver manager. When activating the driver, its `getConnection` method connects to the database and accesses the connection object. Each connection object may participate in only one DB session; therefore, if the program needs to access two or more Linter DBs, each of them must have its own connection object. An application that is to access several DBs at once must either load several drivers or create several connection objects.

Specifics

The argument values correspond to the JDBC 1.01 specification.

java.sql.PreparedStatement

Use

The `java.sql.PreparedStatement` interface is used to execute pre-translated SQL expressions. It works with parametric SQL expressions.

Specifics

The argument values correspond to the JDBC 1.01 specification.

Example

```
import java.sql.*;
import java.io.*;
public class PreparedStatementDemo{
    public static void main (String[ ] args) {
        try {
            Driver d =(Driver)Class.forName("jdbc.LinJdbc.
            LinterDriver").newInstance( );
            String address = "jdbc:Linter:195.98.69.49:1070:local";
// 195.98.69.49 IP address of Linter JDBC server (linapid)
// 1070 socket number
// local The word "local" must appear here or the name
// of the remote Linter node from NODETAB
            String user = "SYSTEM";
            String password = "MANAGER";
            System.out.println("Driver found. Now connecting to
            database ... ");
            Connection con =
            DriverManager.getConnection(address,user,password);
            // CREATE PREPARED STATEMENT
            PreparedStatement prepstmt;
            //DROP/CREATE TABLE
            Statement stmt = con.createStatement( );
            stmt.executeUpdate("create table test2 (a blob, b int);");
            prepstmt = con.prepareStatement("insert into test2
            values ?,?);");
            System.out.println("Prepared statement created");
            int testSize=10*1024;
            byte[ ] buffer= new byte[testSize];
            for(int jj=0;jj<testSize;jj++)
            buffer[jj]=126;
            System.out.println("Array filled");
            // SET BINARY INPUT STREAM
```

```

        ByteArrayInputStream arr=
        New ByteArrayInputStream(buffer);
        prepstmt.setBinaryStream(1, arr , arr.available( ) );
        // SET INT PARAMETER
        prepstmt.setInt(2, 3);
        // EXECUTE PREPARED UPDATE
        int res;
        res = prepstmt.executeUpdate( );
        System.out.println("Insert result= "+res);
        // EXECUTE QUERY
        ResultSet results;
        results = stmt.executeQuery("select * from test2;");
        byte bt[ ];
        DataInputStream dis;
        if( results.next( ) ) {
            System.out.println("Second column value = "+
                results.getInt(2) );

            dis =
            New DataInputStream( results.getBinaryStream(1) );
            bt = new byte[dis.available( ) ];
            System.out.println("Stream with length="+
                dis.available( )+" created ... now reading ");
            dis.read(bt,1,dis.available( ) );
        }
        results.close( );
        try {
            stmt.executeUpdate("drop table test1;");
        }
        catch(SQLException e){
            System.out.println("Error deleting table:"+e.
                getMessage( ) );
        }
        con.close( );
    }
    Catch (Exception e) {
        System.out.println("Caught      :"+e+"      mess      =      "+e.getMessage( )      );
        e.printStackTrace( );
        return;
    }
}
}

```

java.sql.ResultSet

Use

The java.sql.ResultSet interface provides access to the set of strings received on execution of an SQL expression.

Specifics

The argument values correspond to the JDBC 1.01 specification Linter's ResultSet interface contains new methods that simplify working with cursors.

The standard supports only direct cursor movement, which is usually difficult in real-life application development. To reduce that difficulty, Linter has implemented the following methods:

- | | |
|--------------------------------|---|
| boolean first() | move to the first line of the answer |
| boolean previous() | move to the next line |
| boolean absolute(int position) | move to the selected line |
| boolean relative(int shift) | move to the selected line relative to the current cursor position |

boolean last()

move to the last line

These methods are used via the ResultSetEx interface (see example).

Example

```
import java.sql.*;
import jdbc.LinJdbc.*;
public class ResultSetExDemo {
    public static void main (String[ ] args)    {
        try {
            Driver d =(Driver)Class.forName("jdbc.LinJdbc.
            LinterDriver").newInstance( );
            String      address      ="jdbc:Linter:195.98.69.49:1070:local";
// 195.98.69.49      IP address of Linter JDBC server (linapid)
// 1070      socket number
// local      The word "local" must appear here or the name
//      of the remote Linter node from NODETAB
            String user = "SYSTEM";
            String password = "MANAGER";
            Connection con =
            DriverManager.getConnection(address,user,password);
            Statement stmt = con.createStatement( );
            ResultSetEx results;
            stmt.executeUpdate("create table test1 values
            (a int, b char(20) );");
//INSERT
            stmt.executeUpdate("insert into test1 values(1,'First
            string');");
            stmt.executeUpdate("insert into test1 values(2,'Another
            string');");
            stmt.executeUpdate("insert into test1 values(3,'Third
            string');");
            results = (ResultSetEx)stmt.executeQuery("select * from
            test1;");
            System.out.println("ResultSet contains
            "+results.getRowCount( )+" rows");
            while ( results.next( ) ) {
                // Loop through each column, getting the column
                // data and displaying them
                System.out.print("| "+results.getInt(1));;
                System.out.println("| "+results.getString(2)+" |");
            }
            System.out.println( );
            System.out.println("To previous...");
            if(results.previous( ) ) {
                System.out.print("| "+results.getInt(1));
                System.out.println("| "+results.getString(2)+" |");
            }
            System.out.println("To (current+1)...");
            if(results.relative(1) ) {
                System.out.print("| "+results.getInt(1));
                System.out.println("| "+results.getString(2)+" |");
            }
            System.out.println("To first...");
            if(results.absolute(1) ) {
                System.out.print("| "+results.getInt(1));
                System.out.println("| "+results.getString(2)+" |");
            }
        }
        results.close( );
        try{
            stmt.executeUpdate("drop table test1;");
        }
        Catch (SQLException e) {
            System.out.println("Error deleting table:"+e.getMessage( ) );    }
        con.close( );
    } Catch (Exception e) {
        System.out.println("Caught      :"+e+"      mess      =      "+e.getMessage(      )      );
        e.printStackTrace( );
    }
}
```

```

return;
}
}
}

```

java.sql.Statement

Use

The `java.sql.Statement` is used to send SQL queries to a DB. An SQL query includes not only the text of the query but also such characteristics as parameters and query status.

Specifics

The argument values correspond to the JDBC 1.01 specification.

Example

```

import java.sql.*;
import jdbc.LinJdbc.*;
public class StatementDemo {
    public static void main (String[ ] args) {
        try {
            Driver d =(Driver)Class.forName("jdbc.LinJdbc.
LinterDriver").newInstance( );
            String address = "jdbc:Linter:195.98.69.49:1070:local"; //
195.98.69.49 IP address of Linter JDBC server (linapid)
            // 1070 socket number
            // local The word "local" must appear here or the name
            // of the remote Linter node from NODETAB
            String user = "SYSTEM"; String password = "MANAGER";
            System.out.println("Driver found. Now connecting to
database ... "); Connection con =
            DriverManager.getConnection(address,user,password);
            System.out.println(" Connection established ... "); Statement
stmt = con.createStatement( ); System.out.println("
Statement created ... "); ResultSet results;
            //DROP/CREATE TABLE stmt.executeUpdate("create table test1 (a int, b
char(20) );");
            //INSERT
            stmt.executeUpdate("insert into test1 values(1,'First string');");
            stmt.executeUpdate("insert into test1 values(2,'Another
string');");
            results = stmt.executeQuery("select * from test1;");
            System.out.println(" Result set retrieved... ");
            while ( results.next( ) ) {
                // Loop through each column, getting the column
                // data and displaying
                System.out.print("| "+results.getInt(1));
                System.out.println("| "+results.getString(2)+" |");
            }
            results.close( );
            try{
            stmt.executeUpdate("drop table test1;");
            }
            catch(SQLException e){
                System.out.println("Error deleting table:"+e.
getMessage( ) );
            }
        }
        con.close( );
    }
    Catch (Exception e) {
        System.out.println("Caught :"+e+" mess
="+e.getMessage( ) );
        e.printStackTrace( );
        return;
    }
}
}

```

java.sql.DatabaseMetaData

Use

The `java.sql.DatabaseMetaData` queries the DB for its structure information. It is used when developing applications that will have no information about the database they will be accessing. The `java.sql.DatabaseMetaData` interface allows applications to extract needed information in real-time under on the condition that the user has the right to do so. This allows the application to acquire small but important details, eg., which symbol is used as a terminator and how the DBMS processes NULL values during sorting.

Specifics

The argument values correspond to the JDBC 1.01 specification

java.sql.ResultSetMetaData

Use

The `java.sql.ResultSetMetaData` interface permits extraction of information about data types and column properties in the `ResultSet`. This is especially important in building dynamic systems, such as applications for development and implementation of queries when the DB and its structure are initially unknown.

Specifics

The argument values correspond to the JDBC 1.01 specification.

Standard Classes

The classes listed below are part of the JDBC standard. The Linter JDBC driver developers have not made any modifications to those classes. They may be used freely as part of the JDK distribution package:

`java.sql.Time`

`java.sql.Timestamp`

`java.sql.Types`

`java.sql.SQLException`

`java.sql.SQLWarning`

Driver Installation and Launch

Driver Installation

To install the driver perform the following:

1. Extract the jclasses.zip into a user-defined directory and add the path to the directory to the CLASSPATH variable.
2. After the driver has been included in the Linter distribution package, driver installation will be performed automatically if chosen as one of the installation components.

Driver Launch

Launching the Client Driver

The client side of the driver is launched automatically on the client machine during the driver registration command by an applet or an independent application.

Launching the Server Driver

The linapi (linapid.exe) program must be executed on the server. This program receives queries from the client driver and sends them to the Linter kernel.

The linapi options are:

- `/P= | /P` port address of the server on which the server driver is to be <port> launched (default 1070).
- `/C= | /C` path to the dbhandler directory. The server part of the driver <path> consists of two executable modules - linapi and dbhandler(DBHandler.exe). If dbhandler is located in a directory other than that of linapi, then /C specifies the path to dbhandler.
- `/H= | /?` display information on program options.