JDBC

Java database connectivity

Prof.Trupesh Patel

Points to be discussed

- JDBC: Components of JDBC;
- JDBC Architecture;
- JDBC Drivers,
- CRUD operation Using JDBC and java.sql package,
- DriverManager Class,
- Driver, Connection, Statement and Resultset Interfaces,
- difference between java.sql and javax.sql

JDBC

- JDBC is a specification from sun microsystem that provides a standard abstraction(API/Protocol) for java applications to communicate with different databases.
- Capable to access databases and spreadsheets.
- Platform independent interface between a relation database and the java programming language.

JDBC - Characteristics

- Supports a wide level of portability.
- Provide java interface compatible with java applications.
- Provide higher level API for application programmers.
- Provide JDBC API for java applications.

JDBC - Components

- The JDBC API
- The JDBC DriverManager
- The JDBC test suite
- The JDBC-ODBC bridge

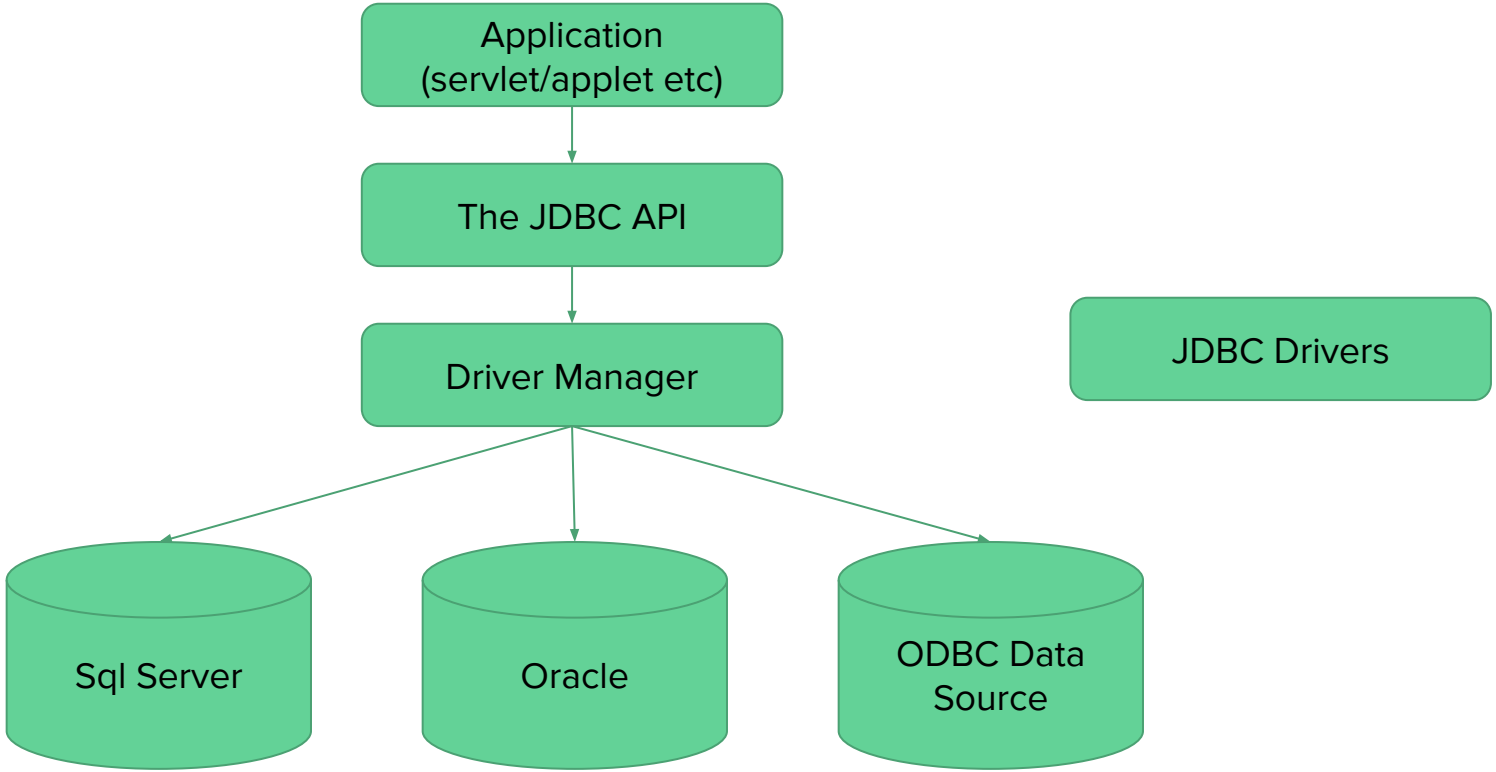
JDBC - Specification

- JDBC 1.0 - basic functionality
- JDBC 2.0 - CORE API and optional package API
- JDBC 3.0 - performance optimization , connection pooling and statement (3.0 is a combination of 2.0 core and optional package)
- JDBC 4.0 - Auto loading of driver interface, connection management, ROWID data type support,Annotation in SQL queries, National character set conversion support, Enhancement to exception handling, Enhanced support for large objects

JDBC - Architecture

- Classes and interfaces in JDBC API :
  - DriverManager
  - Driver
  - Connection
  - Statement
  - PreparedStatement
  - CallableStatement
  - ResultSet
  - DatabaseMetaData
  - ResultSetMetaData
  - SqlData
  - Blob
  - Clob

JDBC - Architecture



JDBC - Drivers

- Type - 1 Driver
- Type - 2 Driver
- Type - 3 Driver
- Type - 4 Driver

CRUD operation using JDBC and java.sql package

MysqlCon.java

```
1   import java.sql.*;
2   class MysqlCon{
3   public static void main(String args[]){
4   try{
5       System.out.println("Testing");
6       Class.forName("com.mysql.jdbc.Driver");
7         Connection con=DriverManager.getConnection(
8         "jdbc:mysql://localhost:3308/surveyapplication","root","password");
9         //here sonoo is database name, root is username and password
10        System.out.println("Demo");
11        Statement stmt=con.createStatement();
12        ResultSet rs=stmt.executeQuery("select * from login_data");
13        while(rs.next())
14        System.out.println(rs.getString(1)+"   "+rs.getString(2)+"   "+rs.getString(3));
15        con.close();
16  }catch(Exception e){ System.out.println(e);}
17  }
18  }
```

Snipped

DriverManager class

- It is a non abstract class in JDBC API.
- Maintains a list of DriverInfo objects, where each driverinfo object holds one driver implementation class object and its name.
- Prepares a connection using the driver implementation that accepts the given JDBC URL.

DriverManager class

Modifier and Type	Method and Description
`static void`	`deregisterDriver(Driver driver)` Removes the specified driver from the `DriverManager`'s list of registered drivers.
`static Connection`	`getConnection(String url)` Attempts to establish a connection to the given database URL.
`static Connection`	`getConnection(String url, Properties info)` Attempts to establish a connection to the given database URL.
`static Connection`	`getConnection(String url, String user, String password)` Attempts to establish a connection to the given database URL.
`static Driver`	`getDriver(String url)` Attempts to locate a driver that understands the given URL.

DriverManager class

`static Enumeration<Driver>`	**`getDrivers()`** Retrieves an Enumeration with all of the currently loaded JDBC drivers to which the current caller has access.
`static int`	**`getLoginTimeout()`** Gets the maximum time in seconds that a driver can wait when attempting to log in to a database.
`static PrintStream`	**`getLogStream()`** **Deprecated.** Use `getLogWriter`
`static PrintWriter`	**`getLogWriter()`** Retrieves the log writer.
`static void`	**`println(String message)`** Prints a message to the current JDBC log stream.
`static void`	**`registerDriver(Driver driver)`** Registers the given driver with the `DriverManager`.
`static void`	**`registerDriver(Driver driver, DriverAction da)`** Registers the given driver with the `DriverManager`.
`static void`	**`setLoginTimeout(int seconds)`** Sets the maximum time in seconds that a driver will wait while attempting to connect to a database once the driver has been identified.

DriverManager class

static void	setLogStream(PrintStream out) **Deprecated.** Use setLogWriter
static void	setLogWriter(PrintWriter out) Sets the logging/tracing PrintWriter object that is used by the DriverManager and all drivers.

Driver interface

It is used to create connection object that provide an entry point for database connectivity.

Generally,all drivers provide the DriverManager class that implements the Driver interface and helps to load the driver in a JDBC application.

The drivers are loaded for any given connection request with the help of the DriverManager class.

After the Driver interface is loaded , its instance is created and registered with the DriverManager Class.

Driver interface

Boolean acceptsURL(string url)	Checks whether or not the specified URL format is acceptable by the driver.
Connection connect(string url,properties info)	Establishes the connection with the database by using given URL.

Connection Interfaces

- Its an abstraction to access the session established with a database server.
- JDBC driver provider must implement the connection interface.
- The connection type of object represents the session established with the data store.

Statement createStatement()	Creates the statement object to send SQL statements to the specified database.
Void close()	Closes a connection and releases the connection object associated with the connected database.
CallableStatement prepareCall(String sql)	Create a callableStatement object to call database stored procedures.
PreparedStatement prepareStatement(String sql)	Create a PreparedStatement object to send the SQL statements over a connection.

Statement Interface

It is an abstraction to execute the SQL statements requested by a user and return the results by using the ResultSet object.

The statement object contains a single ResultSet object at a time.

The statement interface provides specific methods to execute and retrieve the results from a database.

The PreparedStatement interface provides the methods to deal with the IN parameters, the CallableStatement interface provides methods to deal with IN and OUT parameters.

Statement Interface

Void addBatch(String sql)	Adds the SQL commands to the existing list of commands for the statement object. These commands are executed in a batch by calling the executeBatch() method.
Void abort()	Terminates an open connection with the database.
ResultSet executeQuery(String sql)	Execute the SQL command and returns a single ResultSet object.
Int executeUpdate(String sql)	Execute the sql DDL statements.
Connection getConnection()	Retrieves an object of Connection type ,which is used to maintain the connection of a java application with a database.

JDBC process with java.sql package

Prepared statements

PreparedStatementExample.java

```
1   import java.sql.*;
2   class PreparedStatementExample{
3   public static void main(String args[]){
4   try{
5   Class.forName("com.mysql.jdbc.Driver");
6
7   Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3308/surveyapplication","root","password");
8
9   PreparedStatement stmt=con.prepareStatement("insert into login_data values(?,?,?)");
10  stmt.setInt(1,101);//1 specifies the first parameter in the query
11  stmt.setString(2,"Ratan");
12  stmt.setString(3,"Ratan123");
13
14  int i=stmt.executeUpdate();
15  System.out.println(i+" records inserted");
16  ResultSet rs=stmt.executeQuery("select * from login_data");
17  while(rs.next())
18  System.out.println(rs.getString(1)+"  "+rs.getString(2)+"  "+rs.getString(3));
19  //con.close();
20  con.close();
21
22  }catch(Exception e){ System.out.println(e);}
23
24  }
25  }
```

Snipped

Prepared statements

PreparedStatementExample.java

```
1    import java.sql.*;
2    import java.util.Properties;
3
4    public class PreparedStatementExample {
5
6        public static void main(java.lang.String[] args)
7        {
8            // Load the following from a properties object.
9            String DRIVER = "com.ibm.db2.jdbc.app.DB2Driver";
10           String URL    = "jdbc:db2://*local";
11
12           // Register the native JDBC driver. If the driver cannot
13           // be registered, the test cannot continue.
14           try {
15               Class.forName(DRIVER);
16           } catch (Exception e) {
17               System.out.println("Driver failed to register.");
18               System.out.println(e.getMessage());
19               System.exit(1);
20           }
21
22           Connection c = null;
23           Statement s = null;
24
25           //    This program creates a table that is
26           //    used by prepared statements later.
27           try {
28               // Create the connection properties.
29               Properties properties = new Properties ();
30               properties.put ("user", "userid");
31               properties.put ("password", "password");
32
33               // Connect to the local database.
34               c = DriverManager.getConnection(URL, properties);
```

Prepared statements

```
30              properties.put ("user", "userid");
31              properties.put ("password", "password");
32
33              // Connect to the local database.
34              c = DriverManager.getConnection(URL, properties);
35
36              // Create a Statement object.
37              s = c.createStatement();
38              // Delete the test table if it exists.  Note that
39              // this example assumes throughout that the collection
40              // MYLIBRARY exists on the system.
41              try {
42                  s.executeUpdate("DROP TABLE MYLIBRARY.MYTABLE");
43              } catch (SQLException e) {
44                  // Just continue... the table probably did not exist.
45              }
46
47              // Run an SQL statement that creates a table in the database.
48              s.executeUpdate("CREATE TABLE MYLIBRARY.MYTABLE (NAME VARCHAR(20), ID INTEGER)");
49
50          } catch (SQLException sqle) {
51              System.out.println("Database processing has failed.");
52              System.out.println("Reason: " + sqle.getMessage());
53          } finally {
54              // Close database resources
55              try {
56                  if (s != null) {
57                      s.close();
58                  }
59              } catch (SQLException e) {
60                  System.out.println("Cleanup failed to close Statement.");
61              }
62          }
63
```

Prepared statements

```
65          //    This program then uses a prepared statement to insert many
66          //    rows into the database.
67          PreparedStatement ps = null;
68          String[] nameArray = {"Rich", "Fred", "Mark", "Scott", "Jason",
69              "John", "Jessica", "Blair", "Erica", "Barb"};
70          try {
71              // Create a PreparedStatement object that is used to insert data into the
72              // table.
73              ps = c.prepareStatement("INSERT INTO MYLIBRARY.MYTABLE (NAME, ID) VALUES (?, ?)");
74
75              for (int i = 0; i < nameArray.length; i++) {
76                  ps.setString(1, nameArray[i]);      // Set the Name from our array.
77                  ps.setInt(2, i+1);                  // Set the ID.
78                  ps.executeUpdate();
79              }
80
81          } catch (SQLException sqle) {
82              System.out.println("Database processing has failed.");
83              System.out.println("Reason: " + sqle.getMessage());
84          } finally {
85              // Close database resources
86              try {
87                  if (ps != null) {
88                      ps.close();
89                  }
90              } catch (SQLException e) {
91                  System.out.println("Cleanup failed to close Statement.");
92              }
93          }
94
```

Prepared statements

```
92              }
93          }
94
95
96          //    Use a prepared statement to query the database
97          //    table that has been created and return data from it. In
98          //    this example, the parameter used is arbitrarily set to
99          //    5, meaning return all rows where the ID field is less than
100         //    or equal to 5.
101         try {
102             ps = c.prepareStatement("SELECT * FROM MYLIBRARY.MYTABLE " +
103                                     "WHERE ID <= ?");
104
105             ps.setInt(1, 5);
106
107             // Run an SQL query on the table.
108             ResultSet rs = ps.executeQuery();
109             // Display all the data in the table.
110             while (rs.next()) {
111                 System.out.println("Employee " + rs.getString(1) + " has ID " + rs.getInt(2));
112             }
113
114         } catch (SQLException sqle) {
115             System.out.println("Database processing has failed.");
116             System.out.println("Reason: " + sqle.getMessage());
117         } finally {
118             // Close database resources
119             try {
120                 if (ps != null) {
121                     ps.close();
122                 }
123             } catch (SQLException e) {
124                 System.out.println("Cleanup failed to close Statement.");
125             }
126
127             try {
128                 if (c != null) {
129                     c.close();
130                 }
131             } catch (SQLException e) {
132                 System.out.println("Cleanup failed to close Connection.");
133             }
134
135         }
136     }
137 }
```

Callable statements

CallableStatement interface is used to call the **stored procedures and functions**.

We can have business logic on the database by the use of stored procedures and functions that will make the performance better because these are precompiled.

Suppose you need the get the age of the employee based on the date of birth, you may create a function that receives date as the input and returns age of the employee as the output.

Callable statements

How to get the instance of CallableStatement?

The prepareCall() method of Connection interface returns the instance of CallableStatement. Syntax is given below:

```
1. public CallableStatement prepareCall("{ call procedurename(?,?...?)}");
```

The example to get the instance of CallableStatement is given below:

```
1. CallableStatement stmt=con.prepareCall("{call myprocedure(?,?)}");
```

It calls the procedure myprocedure that receives 2 arguments.

Callable statements

A procedure (often called a stored procedure) is a **collection of pre-compiled SQL statements** stored inside the database.

Stored Procedure Features

- Stored Procedure increases the performance of the applications. Once stored procedures are created, they are compiled and stored in the database.
- Stored procedure reduces the traffic between application and database server. Because the application has to send only the stored procedure's name and parameters instead of sending multiple SQL statements.
- Stored procedures are reusable and transparent to any applications.
- A procedure is always secure. The database administrator can grant permissions to applications that access stored procedures in the database without giving any permissions on the database tables.

Callable statements

Stored procedure syntax

- DELIMITER &&
2. **CREATE PROCEDURE** procedure_name [[IN | **OUT** | INOUT] parameter_name datatype [, parameter datatype])]
3. **BEGIN**
4. Declaration_section
5. Executable_section
6. **END** &&
7. DELIMITER ;

Callable statements

IN parameter

It is the default mode. It takes a parameter as input, such as an attribute. When we define it, the calling program has to pass an argument to the stored procedure. This parameter's value is always protected.

OUT parameters

It is used to pass a parameter as output. Its value can be changed inside the stored procedure, and the changed (new) value is passed back to the calling program. It is noted that a procedure cannot access the OUT parameter's initial value when it starts.

INOUT parameters

It is a combination of IN and OUT parameters. It means the calling program can pass the argument, and the procedure can modify the INOUT parameter, and then passes the new value back to the calling program.

Callable statements

CallableDemo.java

```java
// Java program to use Callable Statement
// in Java to call Stored Procedure

package CallableDemo;

import java.sql.*;

public class CallableDemo {

    public static void main(String[] args) throws Exception
    {
        Class.forName("com.mysql.jdbc.Driver");

        // Getting the connection
        Connection con = DriverManager.getConnection("jdbc:mysql://localhost/root", "acm", "acm");

        String sql_string = "insert into students values(?,?,?)";

        // Preparing a CallableStateement
        CallableStatement cs = con.prepareCall(sql_string);

        cs.setString(1, "geek1");
        cs.setString(2, "python");
        cs.setString(3, "beginner");
        cs.execute();
        System.out.print("uploaded successfully\n");
    }
}
```

Snipped

Callable statements

```
import java.sql.*;
import java.util.ArrayList;
public class Candidate {
   public static ArrayList<String> getSkills(int candidateId) {
      var query = "{ call get_candidate_skill(?) }";
      var skills = new ArrayList<String>();
      try (var conn = MySQLConnection.connect();
           var stmt = conn.prepareCall(query);
           ) {
         stmt.setInt(1, candidateId);
         try(var rs = stmt.executeQuery()){
            while (rs.next()) {
               skills.add(rs.getString("skill"));
            }
         } catch (SQLException e) {
            e.printStackTrace();
         }
      } catch (SQLException e) {
         e.printStackTrace();
      }
      return skills;   } }
```

```
+-----+------------+-----------+-------+
| id  | first_name | last_name | skill |
+-----+------------+-----------+-------+
| 133 | John       | Doe       | Java  |
| 133 | John       | Doe       | JDBC  |
| 133 | John       | Doe       | MySQL |
+-----+------------+-----------+-------+
3 rows in set (0.00 sec)
```

Output

```
Java
JDBC
MySQL
```

Java.sql Vs javax.sql

Provides core JDBC API functionalities.	Provides extended JDBC functionalities.
Focuses on basic database interactions such as connections, statements, and result sets.	ocuses on connection pooling, distributed transactions, and advanced row set features.
`Connection`: Represents a connection to a database. `Statement`, `PreparedStatement`, `CallableStatement`: Represent SQL statements to be executed against the database. `ResultSet`: Represents the result set of a query. `DatabaseMetaData`: Provides metadata about the database. `ResultSetMetaData`: Provides metadata about the result set.	`DataSource`: Represents a factory for connections to a physical data source. `ConnectionPoolDataSource`, `PooledConnection`: Used for connection pooling. `XADataSource`, `XAConnection`: Used for distributed transactions. `RowSet`: Represents a set of tabular data. `RowSetListener`, `RowSetEvent`: Used for event handling in `RowSet` objects. `CachedRowSet`, `JdbcRowSet`, `WebRowSet`: Different types of `RowSet` implementations for handling data in various ways.
`DriverManager`: Manages a list of database drivers. `SQLException`: Exception thrown when a database access error occurs. `Date`, `Time`, `Timestamp`: Used for handling SQL date and time values.	`RowSetMetaDataImpl`: Implements the `RowSetMetaData` interface. `RowSetWarning`: Provides information about warnings that occur on `RowSet` objects.