Unit-4
Database Connectivity

Java Database Connectivity (JDBC)

Java Database Connectivity (JDBC) is an Application Programming Interface (API) used to connect Java application with Database. It is used to interact with various types of Databases such as Oracle, MS Access, My SQL and SQL Server. JDBC is the platform-independent interface.

It allows java program to execute SQL statement and retrieve result from database. The JDBC API consists of classes and methods that are used to perform various operations like: connect, read, write and store data in the database.

We can use JDBC API to handle database using Java program and can perform the following activities:

1. Connect to the database
2. Execute queries and update statements to the database
3. Retrieve the result received from the database.

JDBC Architecture

In general, JDBC Architecture consists of two layers:

- ***JDBC API:*** This provides the application-to-JDBC Manager connection.
- ***JDBC Driver API:*** This supports the JDBC Manager-to-Driver connection.

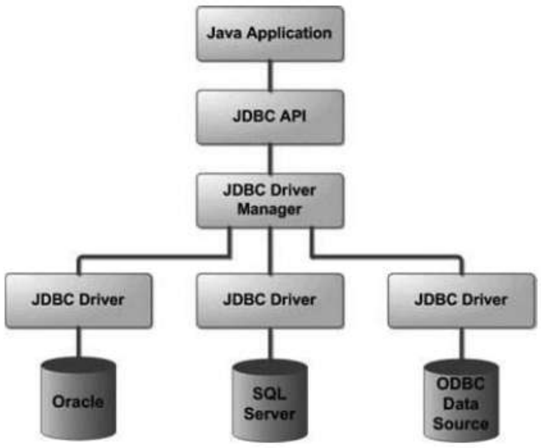


Fig: JDBC Architecture

- ***Java Application*** is a JDBC Client that wants to communicate with the database.
- ***JDBC API*** of an application provides the necessary classes and interfaces to connect the application with the driver manager.
- ***JDBC Driver Manager*** controls interaction between the user-interface and database driver being used.
- ***JDBC Driver*** is a software component that enables java application to interact with the database.
- ***Database Server:*** It is nothing but the Database server like Oracle, MySQL, SQL Server, etc. with which the JDBC client wants to communicate.

Types of JDBC Architecture

There are two types of processing models in JDBC architecture: *two-tier* and *three-tier*.

- ***Two-tier Architecture:*** In the two-tier model, a Java applet or application talks directly to the database. This requires a JDBC driver that can communicate with the particular database management system being accessed. A user's SQL statements are delivered to the database, and the results of those statements are sent back to the user.
- ***Three-tier Architecture:*** In the three-tier model, commands are sent to a "middle tier" of services, which then send SQL statements to the database. The database processes the SQL statements and sends the results back to the middle tier, which then sends them to the user.

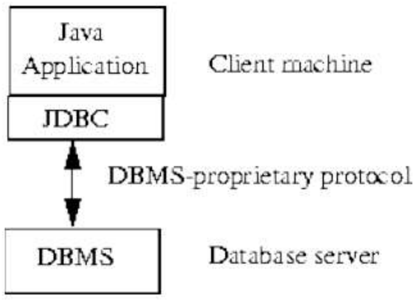
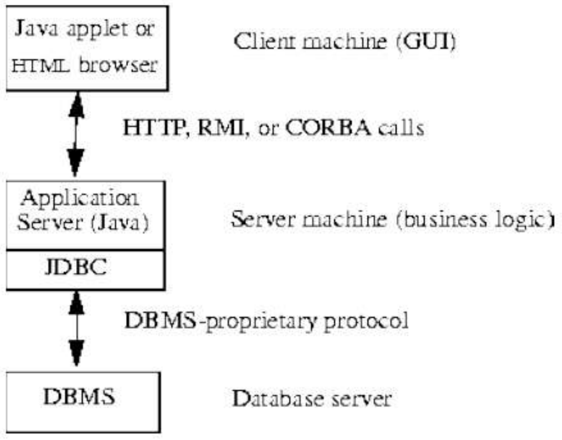


Fig: *Two-tier architecture of JDBC*


Java applet or HTML browser
Client machine (GUI)
HTTP, RMI, or CORBA calls
Application Server (Java)
Server machine (business logic)
JDBC
DBMS-proprietary protocol
DBMS
Database server
Fig: *Three-tier architecture of JDBC*

JDBC Driver Types

JDBC Driver is a software component that enables java application to interact with the database. There are 4 types of JDBC drivers:

- ***Type 1: JDBC-ODBC bridge***
- ***Type 2: Native-API driver***
- ***Type 3: Net pure Java driver***
- ***Type 4: Pure Java driver***

Type 1: JDBC-ODBC Bridge

- To use a Type 1 driver in a client machine, an ODBC driver should be installed and configured correctly.
- This type of driver does not directly interact with the database. To interact with database, it needs ODBC driver.
- The JDBC-ODBC bridge driver converts JDBC method class into ODBC method calls.
- It can be used to connect to any type of the databases.

Type 2: Native –API driver

- Type 2 drivers are written partially in Java and partially in native code.
- The Native-API of each database should be installed in the client system before accessing a particular database. So in this way, a Native-API driver is database specific driver.
- This driver converts JDBC method calls into native calls of the database API.

Type 3: Net Pure Java Driver

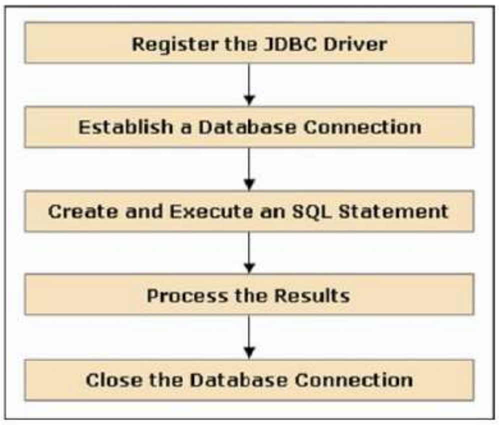
- In a Type 3 driver, the JDBC driver on the client machine uses the socket to communicate with the middleware at the server. The middleware or server acts as a gateway to access the database.
- The client database access requests are sent through the network to the middleware or server and then the middleware converts the request to the database specific API.
- Type-3 drivers are fully written in Java, hence they are portable drivers.

Type 4: Pure Java Driver

- This driver interact directly with database. It does not require any native database library and middleware server that is why it is also known as Thin Driver.
- No client-side or server-side installation.
- It is fully written in Java language, hence they are portable drivers.

Steps to Connect to the Database in Java

Following figure displays the steps to develop a JDBC application:


Register the JDBC Driver
Establish a Database Connection
Create and Execute an SQL Statement
Process the Results
Close the Database Connection

1. ***Register the driver class:*** First step is to load or register the JDBC driver for the database. ***Class*** class provides ***forName()*** method to dynamically load the driver class.
   Syntax: *Class.forName("driverClassName");*

   To load or register OracleDriver class: *Class.forName("oracle.jdbc.driver.OracleDriver");*

RDBMS	JDBC Driver Name

MySQL	Driver Name
	`com.mysql.jdbc.Driver`
	Database URL format:
	`jdbc:mysql//hostname/databaseName`

Oracle	Driver Name:
	`oracle.jdbc.driver.OracleDriver`
	Database URL format:
	`jdbc:oracle:thin@hostname:portnumber:databaseName`

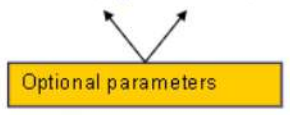
DB2	Driver Name:
	`COM.ibm.db2.jdbc.net.DB2Driver`
	Database URL format:
	`jdbc:db2:hostname:portnumber/databaseName`

Access	Driver Name:
	`com.jdbc.odbc.JdbcOdbcDriver`
	Database URL format:
	`jdbc:odbc:databaseName`

2. ***Making a Connection:*** *DriverManager* class provides the facility to create a connection between a database and the appropriate driver. To open a database connection we can call *getConnection()* method of *DriverManager* class.

   Syntax:

   ```
   Connection connection =
               DriverManager.getConnection(url, username, password);
   ```

   To create a connection with Oracle database:

   Connection connection =
   DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","user","password");

3. ***Creating Statement:*** The statement object is used to execute the query against the database. A statement object can be any one of the *Statement, CallableStatement,* and *PreparedStatement types* .To create a statement object we have to call ***createStatement()*** method of *Connection* interface.

   Syntax: *Statement stmt=connection.createStatement();*

4. ***Executing Statement:*** The *executeQuery()* method of Statement interface is used to execute queries to the database. This method returns the object of *ResultSet* that can be used to get all the records of a table.

   To execute a statement for select query use below:

   ResultSet resultSet = stmt.executeQuery(selectQuery);

   Accessing a ResultSet:
   - ***Cursor operations:*** *first(), last(), next(), previous()*, etc.
   - ***Typical code:***
     ```
     while( rs.next() ) {
         // process the row;
     }
     ```
   - The *ResultSet* class contains many methods for accessing the value of a column of the current row. E.g. *getString(), getDate(), getInt(), getFloat(), getObject()*

   Example:
   ```
   ResultSet rs=stmt.executeQuery("select * from user");
    while(rs.next())
   {
     System.out.println(rs.getString(1)+" "+rs.getString(2));
   }
   ```

5. ***Closing Connection:*** After done with the database connection we have to close it. Use *close()* method of *Connection* interface to close database connection. The statement and ResultSet objects will be closed automatically when we close the connection object.

   Example: *connection.close();*

Complete Example

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class Simplejdbc{
   public static void main( String args[] ) {
      try{
         //Load the JDBC driver
         Class.forName("com.mysql.jdbc.Driver");

         // Establish a connection
         String url = "jdbc:mysql//localhost:3306/test";
         Connection conn = DriverManager.getConnection(url);

         //Create a statement
         Statement st = conn.createStatement();

         //Execute a statement
         ResultSet rs = st.executeQuery("SELECT * FROM Employee");
         while(rs.next()){
            int id = rs.getInt("E_ID");
            String name = rs.getString("E_Name");
            String address = rs.getString("E_Address");
            Date d = rs.getDate("DOB");
            System.out.println(id+"\t"+name+"\t"+address+"\t"+d);
         }

         rs.close();
         st.close();
         conn.close();
      }catch (SQLException sqlExcep){
         System.out.println("Error: " + sqlExcep.getMessage());
            }
   }
}
```

Q. Write a Java program using JDBC to extract name of those students who live in Kathmandu district, assuming that the student table has four attributes (ID, name, district, and age).

Solution:

Before Writing the code please run the Xampp server and create a database name ***test*** and add a table called ***student*** with (id, name, district and age) column name.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
```

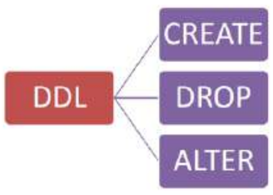
```
public class Studentjdbc{
   public static void main( String args[] ) {
      try{
         //Load the JDBC driver
         Class.forName("com.mysql.jdbc.Driver");

         // Establish a connection
         String url = "jdbc:mysql//localhost:3306/test";
         String username = "root";
         String password = "";
         Connection conn = DriverManager.getConnection(url, username, password);

         //Create a statement
         Statement st = conn.createStatement();

         //Execute a statement
         ResultSet rs = st.executeQuery("SELECT name FROM student WHERE district =
                          'Katmandu' ");

         while(rs.next()){
            String name = rs.getString("name");
            System.out.println("Name: "+ name);
         }

         st.close();
         conn.close();
      }catch (SQLException sqlExcep){
         System.out.println("Error: " + sqlExcep.getMessage());
      }
   }
}
```

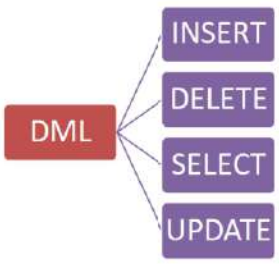
executeQuery() vs. executeUpdate()

- The ***executeQuery()*** method is used to execute a SELECT statement and returns a ResultSet with the number of rows selected.
- The ***executeUpdate()*** is used to execute SQL statements such as INSERT, UPDATE or DELETE and it returns the number of rows affected.

DDL and DML Operations using Java

We use the DDL commands for creating the database or schema, while DML commands are used to populate and manipulate the database. DDL commands can affect the whole database or table, whereas DML statements only affect single or multiple rows based on the condition specified in a query.





Example

```
import java.sql.*;
public class DatabaseExample {
   public static void main(String[] args) {
      try {
         Class.forName("com.mysql.jdbc.Driver");

         String url = "jdbc:mysql://localhost:3306/mydatabase";
         String username = "root";
         String password = "password";
         Connection con = DriverManager.getConnection(url, username, password);

          Statement stmt = con.createStatement();

         // Create a new table using DDL
         String createTableSql = "CREATE TABLE users (id INT PRIMARY KEY, name
                                   VARCHAR(255), email VARCHAR(255))";
         stmt.executeUpdate(createTableSql);

         // Insert data into the table using DML
         String insertSql = "INSERT INTO users VALUES (1, 'Manisha', 'mss@gmail.com')";
         stmt.executeUpdate(insertSql);

         // Retrieve data from the table using DML
         String selectSql = "SELECT * FROM users";
         ResultSet rs = stmt.executeQuery(selectSql);
         while (rs.next()) {
            int id = rs.getInt("id");
            String name = rs.getString("name");
            String email = rs.getString("email");
            System.out.println("ID: " + id + ", Name: " + name + ", Email: " + email);
         }

         // Update data in the table using DML
         String updateSql = "UPDATE users SET name = 'Shila' WHERE id = 1";
         stmt.executeUpdate(updateSql);

         // Delete data from the table using DML
         String deleteSql = "DELETE FROM users WHERE id = 1";
         stmt.executeUpdate(deleteSql);

         // Drop the table using DDL
         String dropTableSql = "DROP TABLE users";
         stmt.executeUpdate(dropTableSql);

         stmt.close();
         con.close();
      }
      catch (SQLException e) {
         e.printStackTrace();
      }
   }
}
```

SQL Exceptions

In JDBC, we may get exceptions when we execute or create the query. SQL exceptions can occur either if the driver is missing or driver name is wrong or information about the database is wrong or the SQL query is wrong. Using Exception handling, we can handle the SQL Exception like we handle the normal exception.

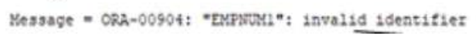
An *SQLException* can occur both in the driver and the database. When such an exception occurs, an object of type *SQLException* will be passed to the catch clause. *SQLException* is available in the *java.sql* package.

The *SQLException* object has the following methods: *getErrorCode( ), getMessage( ), getSQLState( ), getNextException( ), printStackTrace( ).*

Example

```
import java.sql.*;
public class Exception_Example {
   public static void main(String[] args) throws ClassNotFoundException {
      //Update query to set the email id for the employee whose empNUM is 10011
      String update_query = "UPDATE employee_details SET email='jayanta@gmail.com'
                                    where empNum1 = 10011";
      try{
         Class.forName("com.mysql.jdbc.Driver");
         Connection conn =
                  DriverManager.getConnection("jdbc:mysql://localhost:3306/mydatabase");
         Statement statemnt1 = conn.createStatement();
         int return_rows = statemnt1.executeUpdate(update_query);
         System.out.println("No. of Affected Rows = "+ return_rows);
      }
      catch(SQLException sqe)
      {
         System.out.println("Message = " + sqe.getMessage());
      }
   }
}
```

Output:



```
Message = ORA-00904: "EMPNUM1": invalid identifier
```

Prepared Statement

PreparedStatement is used to execute specific queries that are supposed to run repeatedly, for example, *SELECT * from Employees WHERE EMP_ID=?*. This query can be run multiple times to fetch details of different employees. *PreparedStatement* accepts parameterized SQL quires and we can pass 0 or more parameters to this query. Initially this statement uses place holders **"?"** instead of parameters, later on we can pass arguments to these dynamically using the ***setXXX()*** methods of the ***PreparedStatement*** interface, where XXX represents the Java data type of the value we wish to bind the input parameter.

The ***prepareStatement()*** method of *Connection* interface is used to return the object of *PreparedStatement*.

PreparedStatement pstmt = conn.prepareStatement(queryString);

Example:

```
String sql = "select * from people where firstname=? and lastname=?";
PreparedStatement preparedStatement = connection.prepareStatement(sql);
preparedStatement.setString(1, "Jayanta");     // the first parameter is the index of placeholder
preparedStatement.setString(2, "Poudel");
ResultSet result = preparedStatement.executeQuery();
```

<u>Complete Example</u>

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

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

      String mysqlUrl = "jdbc:mysql://localhost/testdb";
      Connection con = DriverManager.getConnection(mysqlUrl, "root", "password");

      String query = "INSERT INTO Employees(Name, RollNo, Location) VALUES (?, ?, ?)";
      PreparedStatement pstmt = con.prepareStatement(query);

      pstmt.setString(1, "Nikita");
      pstmt.setInt(2, 101);
      pstmt.setString(3, "Ramechhap");

      pstmt.setString(1, "Shila");
      pstmt.setInt(2, 102);
      pstmt.setString(3, "Pokhara");

      pstmt.setString(1, "Srishti");
      pstmt.setInt(2, 105);
      pstmt.setString(3, "Gulmi");

      int affectedRows = pstmt.executeUpdate();
      System.out.println(affectedRows + " row(s) affected !!");

      pstmt.close();
      connection.close();
   }
   catch (Exception e) {
     e.printStackTrace();
   }
 }
}
```

Q. You are hired by a reputed software company which is going to design an application for "Movie Rental System". Your responsibility is to design a schema named MRS and create a table named Movie(id, Title, Genre, Language, Length). Write a program to design a GUI form to take input for this table and insert the data into table after clicking the OK button.

Solution:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.sql.*;

public class MovieRentalSystemGUI extends JFrame implements ActionListener {

  JLabel titleLabel, genreLabel, languageLabel, lengthLabel;
  JTextField titleField, genreField, languageField, lengthField;
  JButton okButton;

  public MovieRentalSystemGUI() {
    setTitle("Movie Rental System");
    setLayout(new GridLayout(5, 2, 10, 10));
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setSize(400, 200);

    titleLabel = new JLabel("Title:");
    titleField = new JTextField(20);
    add(titleLabel);
    add(titleField);

    genreLabel = new JLabel("Genre:");
    genreField = new JTextField(20);
    add(genreLabel);
    add(genreField);

    languageLabel = new JLabel("Language:");
    languageField = new JTextField(20);
    add(languageLabel);
    add(languageField);

    lengthLabel = new JLabel("Length (in minutes):");
    lengthField = new JTextField(20);
    add(lengthLabel);
    add(lengthField);

    okButton = new JButton("OK");
    okButton.addActionListener(this);
    add(okButton);

    setVisible(true);
  }
```

```
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == okButton) {
            try {
                Class.forName("com.mysql.jdbc.Driver");
                Connection con =
                 DriverManager.getConnection("jdbc:mysql//localhost:3306/MRS", "root", "root");
                String query = "INSERT INTO Movie (Title, Genre, Language, Length) VALUES (?,
                                                                                        ?, ?, ?)";
                PreparedStatement stmt = con.prepareStatement(query);
                stmt.setString(1, titleField.getText());
                stmt.setString(2, genreField.getText());
                stmt.setString(3, languageField.getText());
                stmt.setInt(4, Integer.parseInt(lengthField.getText()));

                stmt.executeUpdate();
                con.close();
            } catch (SQLException ex) {
                ex.printStackTrace();
            }
        }
    }
    public static void main(String[] args) {
        new MovieRentalSystemGUI();
    }
}
```

Scrollable and Updatable ResultSets

Whenever we create an object of *ResultSet* by default, it allows us to retrieve in the forward direction only and we cannot perform any modifications on *ResultSet* object. Therefore, by default, the *ResultSet* object is non-scrollable and non-updatable ResultSet.

A scrollable updatable result set maintains a cursor which can both scroll and update rows.

Scrollable ResultSet

A scrollable *ResultSet* is one which allows us to retrieve the data in forward direction as well as backward direction but no updations are allowed. To obtain a scrollable result set, we must create a different *Statement* object with the following method:

Statement stmt = conn.createStatement(int resultSetType, int resultSetConcurrency);

Here *resultSetType* represents the type of scrollability and *resultSetConcurrency* represents either read only or updatable. The value of *resultSetType* and the *resultSetConcurrency* are present in *ResultSet* interface as constant data members and they are:

ResultSet Type values:

TYPE_FORWARD_ONLY	The result set is not scrollable (default).
TYPE_SCROLL_INSENSITIVE	The result set is scrollable but not sensitive to database changes.
TYPE_SCROLL_SENSITIVE	The result set is scrollable and sensitive to database changes.

ResultSet Concurrency values:

CONCUR_READ_ONLY	The result set cannot be used to update the database.
CONCUR_UPDATABLE	The result set can be used to update the database.

We can use the following methods to scroll through the result set:

- ***first():*** moves the cursor to the first row.
- ***next():*** moves the cursor forward one row from its current position.
- ***previous():*** moves the cursor to the previous row.
- ***relative(int rows):*** moves the cursor a relative number of rows from its current position. The value of rows can be positive (move forward) or negative (move backward).
- ***absolute(int row):*** moves the cursor to the given row number. The value of row can be positive or negative. A positive number indicates the row number counting from the beginning of the result set. A negative number indicates the row number counting from the end of the result set.

Example

```
import java.sql.*;
class ScrollResultSet {
  public static void main(String[] args) {
     try {
        Class.forName("com.mysql.jdbc.Driver");
        Connection con =
        DriverManager.getConnection("jdbc:mysql//localhost:3306/test", "root", "root");
       Statement st = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                          ResultSet.CONCUR_READ_ONLY);
        ResultSet rs = st.executeQuery("select * from emp");
        System.out.println("RECORDS IN THE TABLE...");
        while (rs.next()) {
           System.out.println(rs.getInt(1) + " " + rs.getString(2));
        }
        rs.first();
        System.out.println("FIRST RECORD...");
        System.out.println(rs.getInt(1) + "     " + rs.getString(2));
        rs.absolute(3);
        System.out.println("THIRD RECORD...");
        System.out.println(rs.getInt(1) + "     " + rs.getString(2));
        rs.last();
        System.out.println("LAST RECORD...");
        System.out.println(rs.getInt(1) + "     " + rs.getString(2));
        rs.previous();
        rs.relative(-1);
        System.out.println("FIRST RECORD...");
        System.out.println(rs.getInt(1) + "     " + rs.getString(2));
        con.close();
     } catch (Exception e) {
        System.out.println(e);
     }
  }
};
```

Updatable ResultSet

An updatable *ResultSet* object allows us to update a column value, insert column values and delete a row. The changes will immediately be persisted in the database and reflected by the *ResultSet* object in real time. In order to make the *ResultSet* object as updatable and scrollable we must use the following constants which are present in *ResultSet* interface.

resultSetType: *TYPE_SCROLL_SENSITIVE*
resultSetConcurrency: *CONCUR_UPDATABLE*

```
Statement st = con.createStatement (ResultSet.TYPE_SCROLL_SENSITIVE,
                                    ResultSet.CONCUR_UPDATABLE);
```

The ***updateXXX()*** method of *ResultSet* interface can be used to change the data in existing row.

Steps for INSERTING a record through ResultSet object

1. Decide at which position we are inserting a record by calling absolute method.
   E.g. *rs.absolute(3);*
2. Since we are inserting a record we must use the following method to make the *ResultSet* object to hold the record.
   rs.moveToInsertRow();
3. Update all columns of the database or provide the values to all columns of database.
   rs.updateXXX(int colno, XXX val);
   E.g. *rs.updateInt(1, 5);*
   rs.updateString(2, "Jayanta");
4. Upto step-3 the data is inserted in *ResultSet* object and whose data must be inserted in the database permanently by calling the following method:
   rs.insertRow();

Steps for UPDATING a record through ResultSet object

1. Decide which record to update.
   E.g. *rs.absolute(3);*
2. Decide which columns to be updated.
   E.g. *rs.updateString(2, "Kushal");*
3. The content of *ResultSet* object must be updated to the database permanently by calling the following method:
   rs.updateRow();

Steps for DELETING a record through ResultSet object

1. Decide which record to delete.
   E.g. *rs.absolute(3);*
2. To delete the record permanently from the database we must call the *deleteRow()* method which is present in ResultSet interface
   rs.deleteRow ();

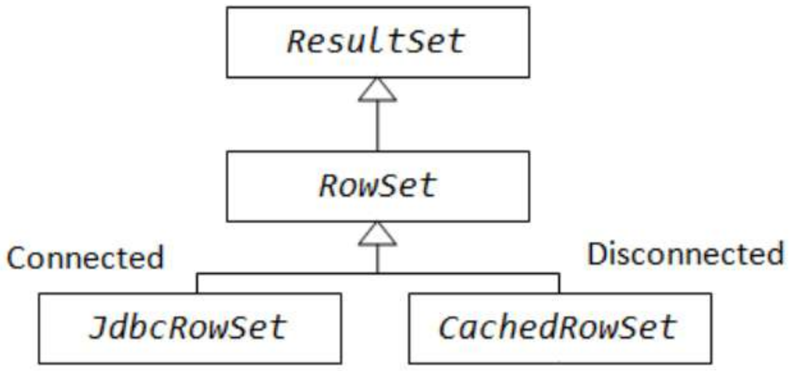
Row Set

Interface *RowSet* provides several *set* methods that allow the programmer to specify the properties needed to establish a connection (such as the database URL, user name, password etc.) and create a *Statement* (such as a query).

There are two types of *RowSet* objects – *connected* and *disconnected.*

- A ***connected RowSet*** object connects to the database once and remains connected until the application terminates.
- A ***disconnected RowSet*** object connects to the database, get data from it, and then close the connection. A program may change the data in a disconnected *RowSet* while it is disconnected. Modified data can be updated to the database after a disconnected *RowSet* reestablishes the connection with the database.

Javax.sql.rowset contains two subinterfaces of RowSet: ***JdbcRowSet*** and ***CachedRowSet.***



- ***JdbcRowSet,*** a connected *RowSet*, acts as a wrapper around a *ResultSet* object, and allows programmer to scroll through and update the rows in the *ResultSet* object. A *JdbcRowSet* object is scrollable and updatable by default.
- ***CachedRowSet,*** a disconnected *RowSet,* caches rows of data in memory and disconnects from the database. It makes possible to operate (scroll and update) without keeping the database connection open all the time. Like *JdbcRowSet*, a *CachedRowSet* object is scrollable and updatable by default. A *CachedRowSet* is also serializable, so it can be passed between Java applications through a network. However, a *CachedRowSet* has a limitation – the amount of data that can be stored in memory is limited.

Example

In this example, *RowSet* is used to retrieve data from database instead of *ResultSet*.

```
import javax.sql.rowset.*;
import java.sql.*;

public class RowSetDemo {

  public static void main(String[] args) {
    String url = "jdbc:mysql://localhost:3306/testDb";
    String userName = "root";
    String password = "root";

    try {

      Class.forName("com.mysql.jdbc.Driver ");
```

```
        // first, create a factory object for rowset
        RowSetFactory rowSetFactory = RowSetProvider.newFactory();

        // create a JDBC rowset from the factory
        JdbcRowSet rowSet = rowSetFactory.createJdbcRowSet();

        // Set connection properties
        rowSet.setUrl(url);
        rowSet.setUsername(userName);
        rowSet.setPassword(password);

        // Set SQL Query to execute
        rowSet.setCommand("SELECT * FROM contact");
        rowSet.execute();
        System.out.println("id \tName \tDepartment \tEmail \t\Salary");

        // Iterating over RowSet
        while (rowSet.next()) {
           System.out.println(rowSet.getInt("id") + "\t"
                 + rowSet.getString("name") + "\t"
                 + rowSet.getString("department") + "\t"
                 + rowSet.getString("email") + "\t"
                 + rowSet.getString("salary"));
        }
     } catch (SQLException sqle) {
        sqle.printStackTrace();
     }
   }
}
```

Executing SQL command by CachedRowSet object

```
String url = "jdbc:mysql://localhost:3306/college";
String username = "root";
String password = "password";

RowSetFactory factory = RowSetProvider.newFactory();
CachedRowSet rowset = factory.createCachedRowSet();

rowset.setUrl(url);
rowset.setUsername(username);
rowset.setPassword(password);

rowset.setCommand(SELECT * FROM Student);
rowset.execute();
```

The following code iterates all rows in the row set and print details of each row:

```
while (rowset.next()) {
        String name = rowset.getString("name");
        String email = rowset.getString("email");
        String major = rowset.getString("major");

        System.out.printf("%s - %s - %s\n", name, email, major);
}
```

Multiple Results

When running a statement that returns more than one result set, we can use the *execute()* method of the Statement class, because it will return a *boolean* value that indicates if the value returned is a result set or an update count. If the *execute()* method returns true, the statement that was run has returned one or more result sets. We can access the first result set by calling the *getResultSet()* method. To determine if more result sets are available, we can call the *getMoreResults()* method, which returns a *boolean* value of true if more result sets are available. If more result sets are available, we can call the *getResultSet()* method again to access them, continuing the process until all result sets have been processed. If the *getMoreResults()* method returns false, there are no more result sets to process.

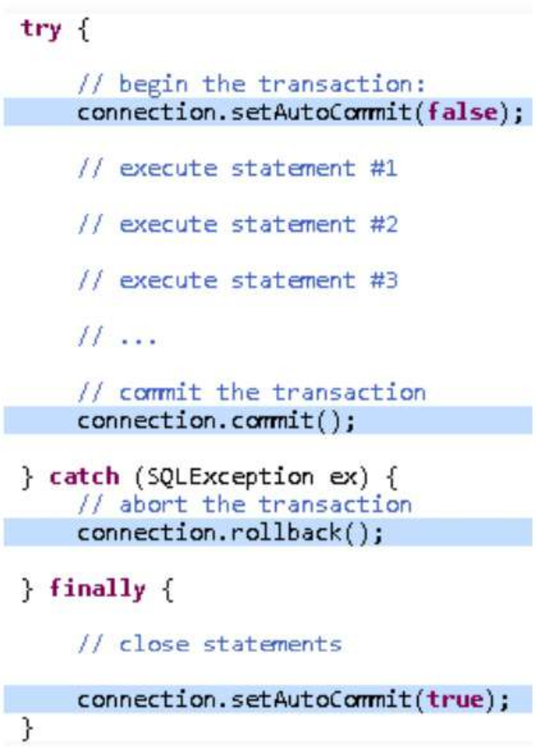
Example

```
import java.sql.*;
public class MultipleResultsExample{
   public static void main(String[] args) {
      String url = "jdbc:mysql://localhost:3306/emp?allowMultipleQueries=true";
      String userName = "root";
      String password = "root";
      try {
            Class.forName("com.mysql.jdbc.Driver ");
            Connection con = DriverManager.getConnection(url, userName, password);
            Statement stmt = con.createStatement();
            int rsCount = 0;
            String SQL = "SELECT * FROM employees WHERE salary<50000; SELECT *
                                                FROM employees WHERE salary>50000";
            boolean results = stmt.execute(SQL);

            do {
               if (results) {
                  ResultSet rs = stmt.getResultSet();
                  rsCount++;

                  System.out.println("RESULT SET #" + rsCount);
                  while (rs.next()) {
                     System.out.println(rs.getString("Id") + ", " + rs.getString("Name") + ", "
                                                            + rs.getString("Salary"));
                  }
               }
               System.out.println();
               results = stmt.getMoreResults();
            } while (results);

            stmt.close();
            con.close();
      } catch (SQLException sqle) {
         sqle.printStackTrace();
      }
   }
}
```

Transactions

A transaction is a set of one or more statements that is executed as a unit, so either all of the statements are executed, or none of the statements is executed. JDBC transaction make sure a set of SQL statements is executed as a unit, either all of the statements are executed successfully, or NONE of the statements are executed (rolled back all changes).

Transactions Handling Workflow in JDBC:



```
try {

    // begin the transaction:
    connection.setAutoCommit(false);

    // execute statement #1

    // execute statement #2

    // execute statement #3

    // ...

    // commit the transaction
    connection.commit();

} catch (SQLException ex) {
    // abort the transaction
    connection.rollback();

} finally {

    // close statements

    connection.setAutoCommit(true);
}
```

- ***<u>Disabling Auto Commit mode</u>:*** By default, a new connection is in auto-commit mode. This means each SQL statement is treated as a transaction and is automatically committed right after it is executed. So we have to disable the auto commit mode to enable two or more statements to be grouped into a transaction: ***conn.setAutoCommit(fasle);***
- ***<u>Committing the transaction</u>:*** After the auto commit mode is disabled, all subsequent SQL statements are included in the current transaction, and they are committed as a single unit until we call the method commit(): ***conn.commit();***

  So a transaction begins right after the auto commit is disabled and ends right after the connection is committed.
- ***<u>Rolling back the transaction</u>:*** If any statement failed to execute, a *SQLException* is thrown, and in the catch block, we invoke the method *rollback()* to abort the transaction:

  conn.rollback();

  Any changes made by the successful statements are discarded and the database is rolled back to the previous state before the transaction.
- ***<u>Enabling Auto Commit mode</u>:*** Finally, we enable the auto commit mode to get the connection back to the default state: ***conn.setAutoCommit(fasle);***

  In the default state (auto commit is enabled), each SQL is treated as a transaction and we don't need to call the *commit()* method manually.

Example

```
import java.sql.*;
class FetchRecords{
public static void main(String args[]){
try{
     Class.forName("oracle.jdbc.driver.OracleDriver");
     Connection conn =
                 DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe",
                                                          "system","oracle");
     conn.setAutoCommit(false);
     Statement stmt=conn.createStatement();
     stmt.executeUpdate("insert into employee values(190,'Jayanta',70000)");
     stmt.executeUpdate("insert into employee values(191,'Arjun',80000)");
     conn.commit();
     stmt.close();
     conn.close();
   }catch(SQlException se){
        se.printStackTrace();
        conn.rollback();
     }
   }
}
```

SQL Escapes

Escape sequences are used within an SQL statement to tell the driver that the escaped part of the SQL string should be handled differently. When the JDBC driver processes the escaped part of an SQL string, it translates that part of the string into SQL code that SQL Server understands.

There are five types of escape sequences that the JDBC API requires, and all are supported by the JDBC driver:

- LIKE wildcard literals
- Function handling
- Date and time literals
- Stored procedure calls
- Outer joins
- Limit escape syntax

LIKE Wildcard Literals

We can specify which escape character to use in strings comparison (with LIKE) to protect wildcards characters ('%' and '_') by adding the following escape : ***{escape 'escape-character'}***. The driver supports this only at the end of the comparison expression.

Example:

Find all rows in which a begins with the character "%"

SELECT a FROM tabA WHERE a LIKE '$%%' {escape '$'}

Find all rows in which a ends with the character "_"

SELECT a FROM tabA WHERE a LIKE '%=_' {escape '='}

Function Handling

The JDBC driver supports function escape sequences in SQL statements with the following syntax:

{fn functionName}

where functionName is a function supported by the JDBC driver. For example:

SELECT {fn UCASE(Name)} FROM Employee

There are various functions that are supported by the JDBC driver when using a function escape sequence:

- ***String Functions:*** ASCII, CONCAT, DIFFERENCE, LENGTH, REPLACE, UCASE, SUBSTRING etc.
- ***Numeric Functions:*** ABS, ACOS, ASIN, ATAN, EXP, LOG, MOD, POWER, SQRT, TRUNCATE etc.
- ***Datetime Functions:*** CURDATE, CURTIME, DAYNAME, HOUR, MINUTE, SECOND,MONTH, WEEK, YEAR etc.
- ***System Functions:*** DATABASE, IFNULL, USER.

Date and time literals

The escape syntax for date, time, and timestamp literals is the following:

{literal-type 'value'}

where *literal-type* is one of the following:

Literal Type	***Description***	***Value Format***
d	*Date*	*yyyy-mm-dd*
t	*Time*	*hh:mm:ss [1]*
ts	*TimeStamp*	*yyyy-mm-dd hh:mm:ss[.f...]*

Example:

UPDATE Orders SET OpenDate={d '2005-01-31'}
WHERE OrderID=1025

Limit escape syntax

The *LIMIT* escape clause can occur in a query at the point where an *OFFSET/FETCH FIRST* clause can appear.

Syntax: *{ LIMIT rowCount [OFFSET startRow] }*

- The *rowCount* is a non-negative integer that specifies the number of rows to return. If *rowCount* is 0, all rows from *startRow* forward are returned.
- The *startRow* is a non-negative number that specifies the number of rows to skip before returning results.

Example:

Return the first two rows of sorted table t

*SELECT * FROM t ORDER BY a { LIMIT 2 }*

Return two rows of sorted table t, starting with the eleventh row

*SELECT * FROM t ORDER BY a { LIMIT 2 OFFSET 10 }*

Please let me know if I missed anything or anything is incorrect.

poudeljayanta99@gmail.com