**Java Database Connectivity:** Introduction, JDBC Architecture, Establishing JDBC Database Connections

JDBC stands for Java Database Connectivity and has been developed by Sun Microsystems. It is a standard Java API that defines how the front-end application (that may be web application or simple Java application) may access the database. It provides a standard library for accessing a wide variety of database systems.

JDBC API and JDBC driver form the important components in order to fetch/store the information to the database. JDBC manager needs a medium in order to communicate with the database. JDBC driver provides this medium and the required information to JDBC manager. JDBC library includes APIs that define interfaces and classes for writing database applications in Java.

Through the JDBC API, we can access a wide variety of database systems including relational as well as non-relational database system.

A diagram of a software development process

Description automatically generated

Management System (RDBMS). It is a database system that is based on relational model. Data in RDBMS is stored in the form of database objects called tables. Table is a collection of related data entries and consists of columns and rows. Some of the most widely used relational database systems include Microsoft MySQL server, Oracle, and IBM’s DB2. Any Java-based application can access a relational database. For this, any RDBM system needs to be installed that provides driver conforming to Java Database Connectivity.

**JDBC Architecture**

JDBC API supports both two-tier and three-tier processing models for database access. This implies that a Java application can communicate directly with the database or through a middle-tier element.

**Two-tier Architecture for Data Access**

A diagram of a diagram of a data processing process

Description automatically generated

In this model, Java application communicates directly with the database. For this, JDBC driver is required and it can establish direct communication with the database. Figure 27.2 illustrates two-tier architecture. As can be seen from the figure, both Java application and JDBC API are located at the client machine and the DBMS and database are located at the database server. User sends commands to the database. The commands are processed and the results of these statements are sent to the user. Java application and the database may reside on the same machine. Alternatively, database may be on the server machine, while the Java application may be on the client machine, which may be connected via the network.

**Three-tier Architecture for Data Access**

A diagram of a computer program

Description automatically generated

In this model, user’s commands are first sent to the application server forming the middle tier. Application server containing the JDBC API sends the SQL statements to the database located on the database server.

The commands are processed and the result is sent to the middle tier, which then sends it to the user. Figure 27.3 depicts the basic three-tier model. Middle tier has often been written in languages such as C or C++ that provide the fast performance. However, Java platform has become the standard platform for middle-tier development with the advancements made in the optimizing compilers. These compilers translate Java byte code into an efficient machine specific code. This model is usually common in web applications in which the client tier is implemented in the web browser. Web server forms the middle tier and the database management system runs on database server. Furthermore, this model provides better performance and simplifies the deployment of applications.

**JDBC Connectivity Model and API**

JDBC API enables Java applications to be connected to relational databases through standard API. This makes possible for the user to establish a connection to a database, create SQL or MySQL statements, execute queries in the database, and so on. JDBC API comprises the following interfaces and classes: Driver manager This class manages a list of database drivers. It matches the connection request from the Java application with the database driver using communication sub-protocol. It acts as an interface between the user and the driver and is used to get a Connection object.

A close-up of a box

Description automatically generated

**Driver** : It handles communication with the database server. JDBC drivers are written in Java language in order to connect with the database. JDBC driver is a software component comprising a set of Java classes that provides linkage between Java program running on Java platform and RDBM system that is residing on the operating system. There are basically four different types of JDBC drivers and these implementations vary because of the wide variety of operating systems and hardware platforms available in which Java operates.

**Type 1 JDBC-ODBC bridge driver** Type 1 JDBC driver provides a standard API for accessing SQL on Windows platform. In this type of the driver, JDBC bridge is used to access ODBC drivers installed on the client machine. For using ODBC, Data Source Name (DSN) on the client machine is required to be configured. The driver converts JDBC interface calls into ODBC calls. It is, therefore, the least efficient driver of the four types. These drivers were mostly used in the beginning and now it is usually used for experimental purposes when no other alternative is available.

**Type 2 driver (also known as Native API driver)** In Type 2 driver, Java interface for vendor-specific API is provided and it is implemented in native code. It includes a set of Java classes that make use of Java Native Interface (JNI) and acts as bridge between Java and the native code. JNI is a standard programming interface that enables Java code running in a Java Virtual Machine (JVM) to call and be called by native applications (these include the programs that are specific to a particular hardware and operating system like C/C++). Thus, the driver converts JDBC method calls into native calls of the database API. For using this driver, it is required that RDBMS system must reside in the same host as the client program. The Type 2 driver provides more functionality and performance than Type 1 driver. For using this driver in a distributed environment, it is required that all the classes that operate on the database should reside on the database host system.

**Type 3 driver (also known as Network-Protocol driver)** It is similar to Type 2 driver but in this case, the user accesses the database through TCP/IP connection. The driver sends JDBC interface calls to an intermediate server, which then connects to the database on behalf of the JDBC driver. Type 3 and 4 drivers are preferably used if the program application does not exist on the same host as the database. It requires database-specific coding to be done in the middle tier.

**Type 4 driver (also known as Native-Protocol driver)** Type 4 JDBC driver is completely written in Java, and thus, it is platform independent. The driver converts JDBC calls directly into vendor-specific database protocol. It is installed inside the Java Virtual Machine of the client and most of the JDBC drivers are of Type 4. It provides better performance when compared to Type 1 and 2 drivers as it does not have the overhead of conversion of calls into ODBC or database API calls. However, at the client side, a separate driver is required for each database.

**Establishing JDBC Database Connections**

This section presents the procedure for connecting to the database system. In JDBC, standardized steps are followed in order to connect to a database and execute the queries.

**Load and register the JDBC driver** In order to make a connection with the database through Java application, first load the suitable JDBC driver into JVM.

**Define the connection** **URL** For establishing connection with the database, URL/address of the database needs to be specified. This connection URL indicates server host, port, and database name with which to establish the connection.

**Establish the connection** Network connection to the database can be established using the connection URL, username, and password.

**Create a statement object** After the connection is established, queries can be performed. For that, Statement object is created that enables the developer to send the queries to the database.

**Execute a query or update** On creating the Statement object, next step is to send SQL statements to the database using execute, executeQuery, executeUpdate, or executeBatch methods.

**Process the results** When the database query is executed, a ResultSet is returned. ResultSet maintains the data in the form of tables (rows and columns). The method returns true or false value depending whether the next row is available or not and if the next row is available, it moves to the next row.

**Close the connection** After retrieving the data, the database connection is closed, thus releasing the resources to the database.

1. **Load and Register the JDBC Driver**

JDBC driver is a software component that interacts with the database server. In order to load the driver, the class name of the database driver is specified that automatically creates a driver instance and registers it with JDBC Driver manager. For loading and registering the driver, specify the class name of database driver as:

A close-up of a sign

Description automatically generated

This will automatically create a driver instance and register it with the JDBC Driver manager. The Class. forName method takes a string argument that specifies the fully qualified class name. The driver is loaded only once and need not be loaded for every connection opened. The method call can throw ClassNotFoundException. Therefore, it is always placed inside the try/catch

A screenshot of a computer

Description automatically generated

1. **Defining the Connection URL**

After loading the JDBC driver, URL address of the database need to be specified for connecting with the database. URL identifies the location of the database server and the type of database. Let us consider the following example.

A chat bubbles with text

Description automatically generated with medium confidence

The general format of URL is: protocol:subprotocol://host:port/database. Protocol in the abovementioned case is jdbc and sub-protocol is mysql. Other sub-protocols may include oracle and odbc. Sub-protocol identifies the type of database, which is mysql in the given example. Default host and port values are localhost and 3306 and can be omitted from the URL.

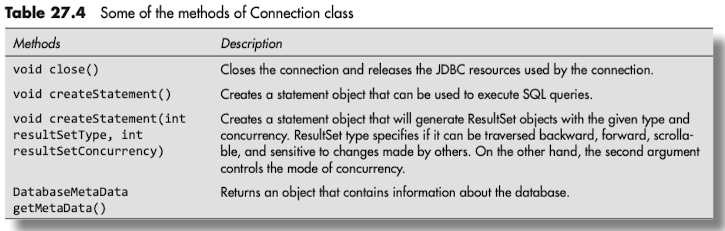
1. **Establishing the connection**

In order to establish the network connection, database URL, username, and password is passed as an argument to getConnection() method of DriverManager class. For example:

A computer code with text

Description automatically generated with medium confidence

Since getConnection() method throws SQLException, it is included within the try/catch block. Connection class comprises various methods.



**A screenshot of a computer

Description automatically generated**

1. **Creating a statement object After**

After the connection is established, a statement object is created to send queries to the database. For this, Connection object’s createStatement() method is used as:

A close-up of a computer code

Description automatically generated

where st is the Statement object.

1. **Closing statement** **object** It is a good practice to close the statement object in order to save the database resources. For this, close() method is used as:

st.close();

1. **Executing a query or update** After getting the Statement object, we need to execute the queries. This is shown in the example:

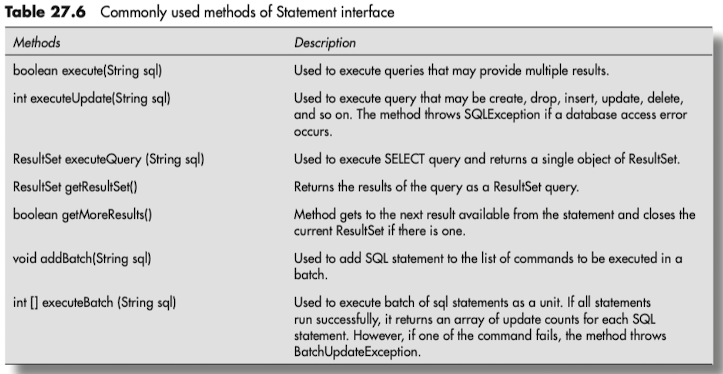
**A screenshot of a computer

Description automatically generated**

**Statement interface** It provides the methods for executing the queries with the database. A Statement object is used for executing static SQL statements and for obtaining the results produced by it. By default, only one ResultSet object is associated with one Statement object. In order to use Statement interface, an object of Statement is first created by calling createStatement() method form java.sql.Connection interface. Its syntax is:

Statement st = con.createStatement();

where st is the object of Statement and con is the object of Connection. After creating object of Statement type, either of the methods executeUpdate() or executeQuery() is invoked. If we are updating the database (i.e., using insert, update, delete, or create), then executeUpdate() method is called. On the other hand, if the Statement object represents an SQL query that returns a ResultSet object, then the method executeQuery() is used. Some of the important methods of Statement interface are given in Table 27.6.



**PreparedStatement interface** It is a sub-interface of Statement. It gives us the flexibility as we can provide the arguments dynamically and execute parameterized query. It provides the advantage of improved performance of JDBC applications as the query is compiled only once. However, it may not always execute faster than the ordinary SQL statement as the performance may also depend upon a particular SQL command that is being executed. Further, it not only improves the performance, but also provides security advantage. It is recommended to use prepareStatement() method to update database values when the input is accepted from the user through an HTML form. This is because it reduces the risk of database being accessed or modified by the attacker. This security risk is also referred to as SQL injection attack.

**Creating the PreparedStatement object** prepareStatement() method defined in the Connection interface is used to create PreparedStatement object. Following code is used to create a PreparedStatement object:

PreparedStatement ps = con.prepareStatement(sql);

Dynamic SQL statements contain input parameters that are required to be passed into the database. For such statements, a position for each input parameter is defined in the query string using the question mark ‘?’, which is also called as the IN parameter. It is a parameter whose value is unknown when the SQL statement is created and it informs the database that a variable is expected in that position. The values are binded to IN parameter with setXXX() method call and the following code shows the syntax:

setXXX(int position, data\_type value);

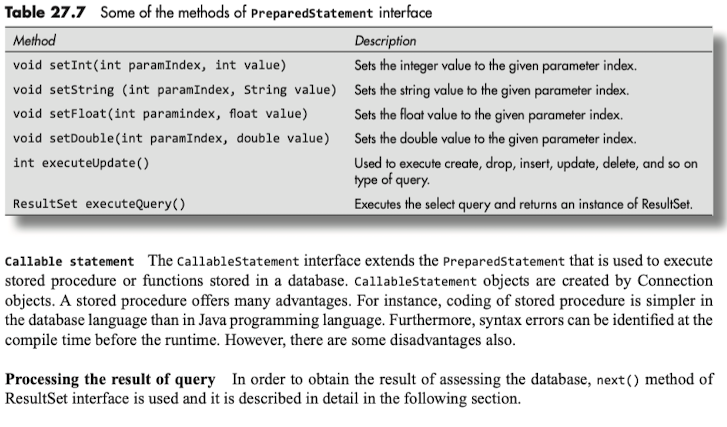
where XXX indicates the associated value type to be assigned, position is an integer that indicates the relative position of the IN parameter in the SQL statement, and value is the actual data value to be assigned to the IN parameter. An example using the setXXX() method is:

String sql = “Delete From Ec\_student WHERE id = ?”;

PreparedStatement ps = con.prepareStatement(sql);

ps.setInt(1,346);

Then, executeQuery() method or executeUpdate() method is used to modify the table data. This is shown in Program 27.7. Some of the important methods of PreparedStatement interface are provided in Table 27.7.



**ResultSet Interface**

ResultSet consists of records. Each record comprises a set of columns. A ResultSet can be created by executing a Statement or PreparedStatement as shown:

Statement st;

st = con.createStatement();

ResultSet rs = st.executeQuery(“select \* from Ec\_Student”);

Using PreparedStatement, ResultSet can be created as:

String sql = select \* from Ec\_Student”;

PreparedStatement st = con.prepareStatement(sql);

ResultSet rs = st.executeQuery();

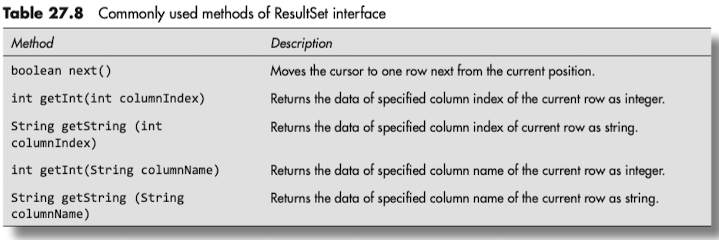
The next() method of ResultSet is used to move through the table one row at a time. It maintains the cursor pointing to one particular row of data (current row at a time). Initially, the cursor is positioned before the first record. The next() method returns true, if the ResultSet has a next record and it moves the ResultSet to point to the next record. In case there are no further records, next() returns false.

Following syntax is used for iterating a ResultSet using the next() method:

A screenshot of a computer

Description automatically generated

The abovementioned code shows that the next() method is actually called before the first record is accessed. When next() returns false, it implies that ResultSet is actually pointing after the last record. While iterating the ResultSet, if column values are required to be accessed, ResultSet provides various getXXX() methods that takes column name or column index as an argument. Methods defined in ResultSet interface are called to move to a particular row and then the method is called to the individual column value. Some of these commonly used methods are described in Table 27.8.



It can be observed from the aforementioned table that the values can be retrieved using either the index number of the column or through the name of the column. In order to retrieve the column value in a particular table, we use one of the abovementioned ResultSet getXXX() method. The values of the column that we want to access can be obtained by passing the name of the column in getXXX() method. For instance,

rs.getInt(“ID”);

rs.getString(“FirstName”);

where rs is the ResultSet object. Following code fragment shows the use of ResultSet object. In the given code, con is the Connection object, rs represents the ResultSet object, and st is the Statement object.

Statement st = con.CreateStatement();

String sql = “select \* from tablename”;

ResultSet rs = st.executeQuery(sql);

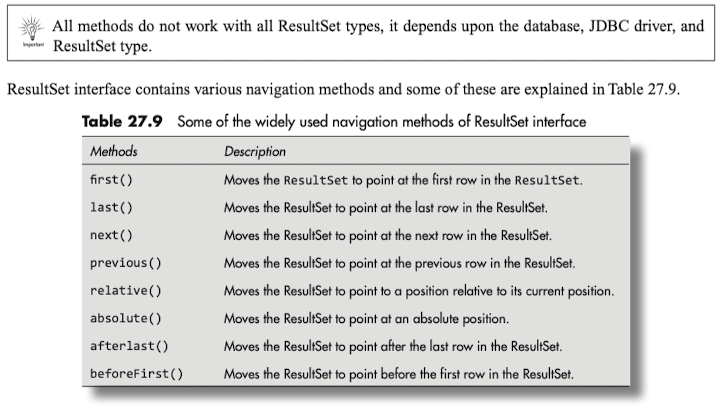
rs.absolute(2);// records of 2nd row is retrieved.

**Navigating the ResultSet**

The cursor can be moved based on the properties of the ResultSet. Following are the properties of ResultSet cursor:

**ResultSet.TYPE\_FORWARD\_ONLY** This is the default type in which the cursor can be moved forward only. **ResultSet.TYPE\_SCROLL\_INSENSITIVE** ResultSet can be navigated (scrolled) both forward and backward. It is also possible to jump to a position relative to the current position or jump to an absolute position. ResultSet is insensitive to changes in the underlying data source, while the ResultSet is open. **ResultSet.TYPE\_ SCROLL\_SENSITIVE** Here, ResultSet can be navigated (scrolled) both forward and backward. In this case, it is possible to jump to a position relative to the current position or jump to an absolute position. However, ResultSet is sensitive to changes in the underlying data source, while the ResultSet is open.

If the property of ResultSet is of the type forward only, then it is possible to navigate through the result set once and in the forward direction only. The scrollable property of ResultSet allows the cursor to be moved both backward and forward in the result set. If the result set is marked as insensitive, then it depicts the corresponding data in the database at the time the result set was created. All the subsequent changes made to the database would not be shown. However, if the result set is marked as sensitive, changes in the database would appear in the result set.



**ResultSetMetaData Interface**

Metadata implies getting data about data, or in other words, getting further information about data. Obtaining metadata about the table involves getting information regarding names of the columns, total number of columns, and column type. ResultSetMetaData interface provides methods to get metadata through the ResultSet object. Commonly used methods of ResultSetMetaData interface are provided in Table 27.10.

A screenshot of a data interface

Description automatically generated

**Creating JDBC Application**

**//JDBC application that depicts inserting the records into table**

import java.sql.\*;

public class InsertData

{

public static void main(String[] args) throws SQLException, ClassNotFoundException

{

String url = "jdbc:mysql://localhost:3306/studentdata";

String user = "root";

String pass = "root";

Connection con;

Statement st;

try{

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

// get connection to database

con = DriverManager.getConnection(url, user, pass);

// creating the statement

st = con.createStatement();

// executing the sql query

String sql;

sql = "insert into Ec\_student" + "(id, firstname, lastname, marks)" + "values('349', 'Priya', 'Jaiswal', '57')";

st.executeUpdate(sql);

}

catch(SQLException e)

{

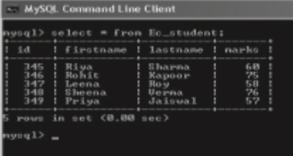
System.err.println(e);

}

}

}

Output:



// JDBC application showing the updation of the records of table

import java.sql.\*;

public class UpdateData

{

public static void main(String[] args) throws SQLException, ClassNotFoundException

{

// JDBC Driver name and database URL

String url = "jdbc:mysql://localhost: 3306/Studentdata";

// Database credentials

String user = "root";

String pass = "root";

Connection con;

Statement st;

try

{

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

// opening the connection

System.out.println("Connecting to Database Studentdata");

con = DriverManager.getConnection(url, user, pass);

System.out.println("Connected database successfuly");

// Executing the query

st = con.createStatement();

String sql = "UPDATE Ec\_student SET marks = '60' where id=349";

st.executeUpdate(sql);

System.out.println("Data updated successfully");

st.close();

con.close();

}

catch(SQLException e)

{

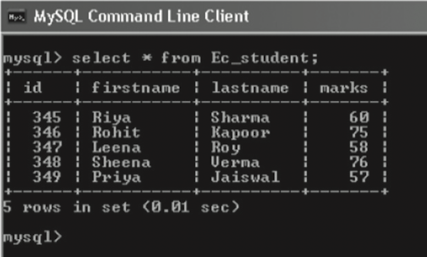
System.err.println(e);

}

}

}

Output



JDBC application for deleting the records of table

import java.sql.\*;

public class DeleteData

{

public static void main(String[] args) throws SQLException, ClassNotFoundException

{

String url = "jdbc:mysql://localhost:3306/studentdata";

// providing database credentials

String user = "root";

String pass = "root";

Connection con;

Statement st;

try

{

// registering the JDBC Driver

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

System.out.println("connecting to database");

con = DriverManager.getConnection(url, user, pass);

System.out.println("connected to database successfully");

// Executing the query

st = con.createStatement();

String sql = "Delete from Ec\_student" + "where id = 345";

st.executeUpdate(sql);

st.close();

con.close();

}

catch(ClassNotFoundException ex)

{

ex.printStackTrace();

} catch(SQLException e)

{

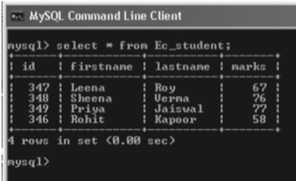
e.printStackTrace();

}

}

}

Output



JDBC application for accessing the records of the table

import java.sql.\*;

public class DisplayData

{

static final String url = "jdbc:mysql://localhost:3306/studentdata";

// providing database credentials

static final String user = "root";

static final String pass = "root";

public static void main(String []args) throws SQLException, ClassNotFoundException

{

Connection con;

Statement st;

ResultSet rs;

try

{ // registering the JDBC Driver

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

con = DriverManager.getConnection(url, user, pass);

System.out. println("connected to database");

// Executing the query

st = con.createStatement();

String sql = "select \* from Ec\_student";

rs = st.executeQuery(sql);

// Extracting the data from result set

while(rs.next())

{

// Retrieving by column name

int id = rs.getInt("id");

String firstname = rs.getString("firstname");

String lastname = rs.getString("lastname");

int marks = rs.getInt("marks");

// Displaying the values

System.out.print("ID:" + id);

System.out.print(", First name:" + firstname);

System.out.print(", Last name:" + lastname);

System.out.print(", Marks:" + marks);

}

rs.close();

st.close();

con.close();

} catch(SQLException e)

{

System.err.println(e);

}

}

}

Output

A white card with black text

Description automatically generated

**Illustration of using some of the methods of ResultSetMetaData interface**

import java.sql.\*;

public class ResultSetDemo

{

static final String url = "jdbc:mysql://localhost:3306/studentdata";

static final String driver = "com.mysql.jdbc.Driver";

// providing database credentials

static final String user = "root";

static final String pass = "root";

public static void main(String[] args)

{

Connection con;

try

{

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

System.out. println("connecting to a selected database");

con = DriverManager.getConnection(url, user, pass);

System.out. println("connected to database");

PreparedStatement ps = con.prepareStatement("select \* from SubjectChoice");

ResultSet rs = ps.executeQuery();

ResultSetMetaData rd = rs.getMetaData();

System.out.println("Total columns: " + rd.getColumnCount());

System.out.println("Column name of first column: " + rd.getColumnName(1));

con.close();

} catch(Exception e)

{

System.out.println(e);

}

}

}

Output

A close up of a text

Description automatically generated