Introduction

JDBC is an API (Application Programming Interface) that helps a programmer to write Java programs to connect to a database, retrieve the data from the database and perform various operations on the data in the Java program.

Every database vendor will provide a document representing all the commands to connect and utilize the database features. This document is called 'API (Application Programming Interface) document'. API document is a file that contains description of all the features of a software, a product or a technology. The vendor of Oracle database has given functions like:

oLog() $\Longrightarrow$ to connect to database

oexec() $\Longrightarrow$ to execute a command

oLogoff() $\Longrightarrow$ to disconnect from Oracle database

In the same way MySQL database has give functions like:

mysql_connect() $\Longrightarrow$ to connect to database

mysql_execute() $\Longrightarrow$ to execute command

mysql_disconnect() $\Longrightarrow$ to disconnect from MySQL database.

This is cumbersome as there are hundereds of functions to learn. This is the reason software scientists have thought of creating a common API document. By using the functions of this common API document, programmers can communicate with any databse in the world, such document is called ODBC (Open Database Connectivity) API.

ODBC is a document that contains common functions to communicate with any database. It is created by Microsoft Corporation. If any organization creates a software depending on this ODBC document, it is called ODBC driver.

In the same way, Sun MicroSystems INC. has also created an API document named JDBC (Java Database Connectivity) API and the actual software which is created according to JDBC API, is called JDBC Driver. JDBC API is defined in java.sql package. This package contains interfaces like `Connection`, `Statement`, `ResultSet`, `CallableStatement`, `Driver`, `ResultSetMetaData`, `PreparedStatement` and classes like `Date`, `Time`, `DriverManager` etc. Several companies have started developing software containing these interfaces and classes. These softwares are called JDBC drivers. For example, **classes12.jar** is a driver developed by Oracle corporation using which we can connect to Oracle database and communicate with it. Similarly **gate.jar** is a JDBC driver from inet to connect to Oracle. Similarly **Mysql-connector-java-3.0.11-stable-bin.jar** is a driver to connect to MySQL database.

Stages in JDBC Program

The following are the stages used by Java Programmers while using `JDBC` in their programs

1. **Registering the Driver:** database driver is a software containing classes and interfaces written according to `JDBC` API. Since there are several drivers avavailable in the market, we should first declare a driver which is going to be used for communication with the database server in a Java Program.
2. **Connecting to a database:** establish a connection with a specific database through the driver which is already registered.
3. **Preparing SQL statement in Java:** create SQL statements in our Java program using any of the interfaces like `Statement`, `PreparedStatement` which are available in java.sql package.
4. **Executing the SQL statements on the database:** `execute()`, `executeQuery()` and `executeUpdate()` methods of Statement interface are used to execute `SQL` statements.
5. **Retrieving the results:** obtained by executing the SQL statements can be stored in an object with the help of interfaces like `ResultSet`, `ResultSetMetaData` and `DatabaseMetaData`.
6. **Closing the connection:** close the connection between the Java program and the database by using `close()` method of `Connection` interface.

Registering the Driver

This is the first step to connect to a database. A programmer should specify which database driver can be used to connect to the database. There are different ways to register a driver.

1. By creating an object to driver class of the driver software. For example to register mysql driver, create an object to the driver class as shown below

   com.mysql.jdbc.Driver = **new** com.mysql.jdbc.Driver();

2. By sending the driver class object to `registerDriver()` method of `DriverManager` class. For example, the object of mysql can be passed to registerDriver method as

   ```
   DriverManager. registerDriver (‘‘ com.mysql.jdbc.Driver’’);
   ```

3. By sending the driver class name directly to `forName()` method as

   ```
   Class.forName(‘‘com.mysql.jdbc.Driver’’);
   ```

Connecting to Database

To connect to database, we should know three things

URL of the database URL (Uniform Resource Locator) represents a protocol to connect to the database. Simply speaking it locates the database on the network.

Username to connect to the database, every user will be given a username which is generally alloted by the database administrator.

Password is the password alloted to the user by the database administrator to connect to the database.

For Example, to connect to mysql database, write the following statement.

```
DriverManager.getConnection
(‘‘ jdbc:mysql://localhost/test ’’, ‘‘ root ’’, ‘‘ root ’’);
```

Here, “jdbc:mysql://localhost/test” is the URL, “root” is the username, “root” is the password.

Example Program to connect to the Database.

```
import java.sql.*;
class ConnectionExample {
    public static void main(String a[]) {
        try {
            Class.forName(‘‘com.mysql.jdbc.Driver’’);
            Connection con = DriverManager.getConnection(‘‘jdbc:mysql://
localhost/db_test ’’, ‘‘ root ’’, ‘‘ root ’’);
            System.out.println(‘‘ Connected...’’);
        }
        catch (SQLException e) {
            System.out.println(‘‘ Exception’’ +e.getMessage());
```

```
            }
            catch (Exception e) {
                System.out. println (''Exception'' +e.getMessage());
            }
        }
}
```

Preparing and Executing SQL Statements

We need SQL (Structured Query Language) statements which are useful to make different operations like creating table in the database, adding data to the database, updating the data of the database, deleting the data from the database and also retrieving the data from the database. SQL statements can be classified into two types as follows.

select statements help to retrieve data from the database in the form of rows.

non-select statements represents all other statements except select statement like create, update, delete etc.

The Statement interface provides three methods for creating SQL queries and returning a result.

boolean execute(sqlstring) returns a boolean indicating if `ResultSet` was returned. Many SQL statements can be executed with execute.

int executeUpdate(sqlstring) returns an int (number of rows affected) or 0 when the statement is a `Data Defination Statement (DDL) statement`, such as `CREATE TABLE`.

ResultSet executeQuery(sqlstring) returns a `ResultSet` object.

For Example, to create a table Student, we can use create statement as:

```
CREATE TABLE Student(sid int(32),
sname Varchar(35), age int(32),
address Varchar(50))
```

To execute the above statement, create statement object, as

```
Statement stmt = con.createStatement();
```

And then, pass the SQL statement to `execute()` method as

```
stmt.execute ('' CREATE TABLE Student(sid int(32),
sname Varchar(35), age int(32),
address Varchar(50))'');
```

Example Program to create table.

```
import java.sql.*;
class CreateTableExample {
        public static void main(String a[]) {
            try {
                Class.forName(''com.mysql.jdbc.Driver'');
                Connection con = DriverManager.getConnection(''jdbc:mysql: //
        localhost/db_test '', ''root '', ''root '');
                Statement stmt = con.createStatement();
                String query = ''CREATE TABLE Student (sid int(32), sname
        Varchar(35), age int(32), address Varchar(50))'';
                stmt.execute(query);
                System.out.println ('' Table Created'');
            }
            catch (SQLException e) {
                System.out.println ('' Exception'' +e.getMessage());
            }
            catch (Exception e) {
                System.out.println ('' Exception'' +e.getMessage());
            }
        }
}
```

Example Program to insert records.

```
import java.sql.*;
class InsertRecordsExample {
        public static void main(String a[]) {
            try {
                Class.forName(''com.mysql.jdbc.Driver'');
                Connection con = DriverManager.getConnection(''jdbc:mysql: //
        localhost/db_test '', ''root '', ''root '');
```

```
            Statement stmt = con.createStatement();
            String query1 = ''INSERT INTO Student VALUES (100, 'Raj', 22, '
    Gachibowli')'';
            String query2 = ''INSERT INTO Student VALUES (101, 'Sagar', 26, '
    Dsnr')'';
            String query3 = ''INSERT INTO Student VALUES (103, 'Sruthi', 20, '
    Vizag')'';
            String query4 = ''INSERT INTO Student VALUES (104, 'Swathi', 28,
    'Mbnr')'';
            stmt.executeUpdate(query1);
            stmt.executeUpdate(query2);
            stmt.executeUpdate(query3);
            stmt.executeUpdate(query4);
            System.out.println(''Records Inserted  Sucessfully'');
        }
        catch (SQLException e) {
            System.out.println(''Exception'' +e.getMessage());
        }
        catch (Exception e) {
            System.out.println(''Exception'' +e.getMessage());
        }
    }
}
```

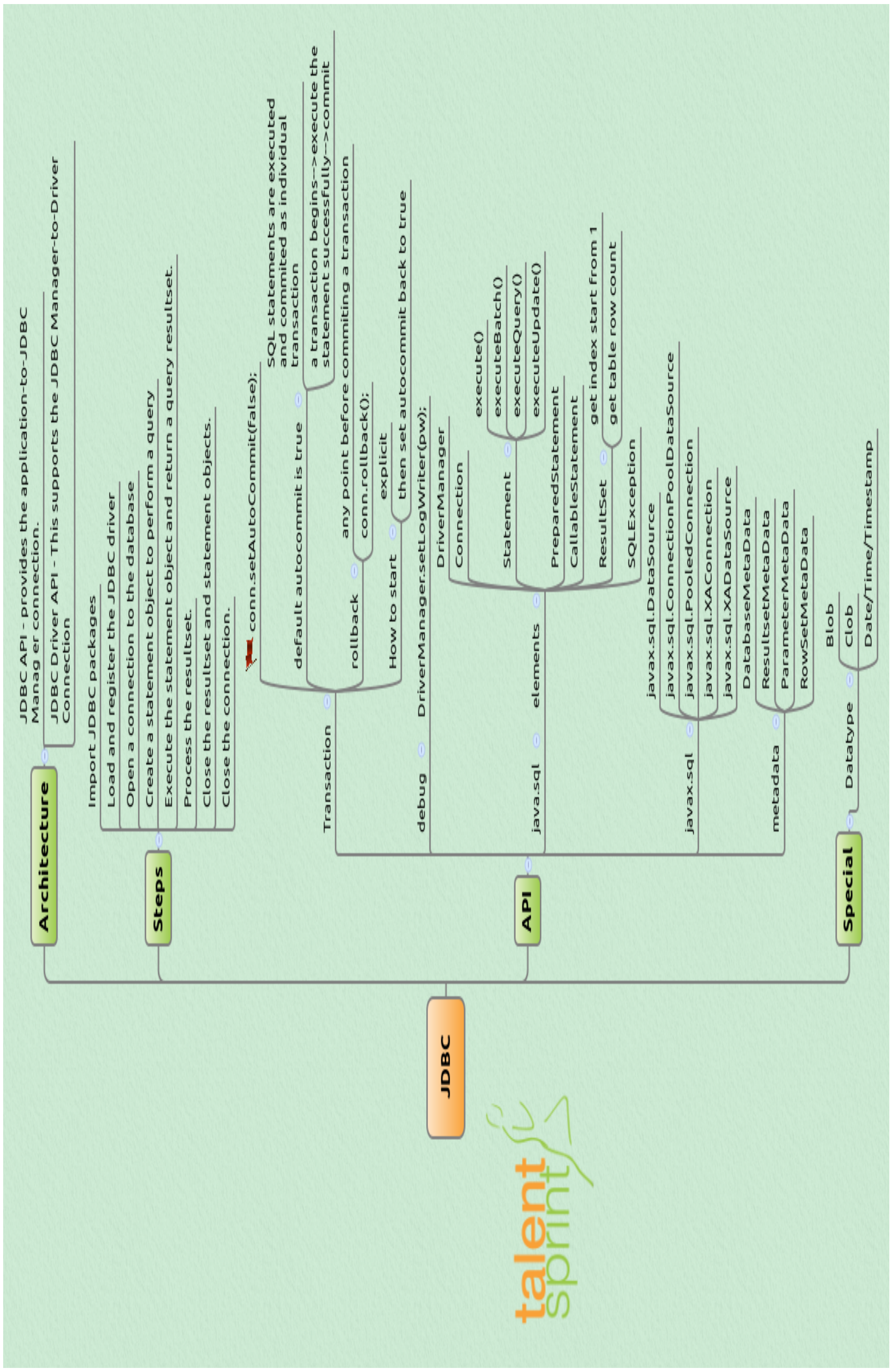
Closing Connection

After executing all the required SQL statements and obtaining the results, close the connection and release the session. This can be done by calling `close()` method of the `Connection` interface as follows.

```
Connection.close();
```

Example Program to Insert records.

```
import java.sql.*;
class InsertCloseExample {
    public static void main(String a[]) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
            Connection con = DriverManager.getConnection("jdbc:mysql://localhost/db_test", "root", "root");
            Statement stmt = con.createStatement();
            String query1 = "INSERT INTO Student VALUES (100, 'Raj', 22, 'Gachibowli')";
            String query2 = "INSERT INTO Student VALUES (101, 'Sagar', 26, 'Dsnr')";
            String query3 = "INSERT INTO Student VALUES (103, 'Sruthi', 20, 'Vizag')";
            String query4 = "INSERT INTO Student VALUES (104, 'Swathi', 28, 'Mbnr')";
            stmt.executeUpdate(query1);
            stmt.executeUpdate(query2);
            stmt.executeUpdate(query3);
            stmt.executeUpdate(query4);
            System.out.println("Records Inserted Sucessfully");
        }
        catch (SQLException e) {
            System.out.println("Exception" +e.getMessage());
        }
        catch (Exception e) {
            System.out.println("Exception" +e.getMessage());
        }
        finally {
            if (con != null) {
                System.out.println("Closing connection");
                con.close();
            } else {
                System.out.println("Connection Closed");
            }
        }
    }
}
```


JDBC
Architecture
JDBC API - provides the application-to-JDBC Manag er connection.
JDBC Driver API - This supports the JDBC Manager-to-Driver Connection
Steps
Import JDBC packages
Load and register the JDBC driver
Open a connection to the database
Create a statement object to perform a query
Execute the statement object and return a query resultset.
Process the resultset.
Close the resultset and statement objects.
Close the connection.
API
Transaction
conn.setAutoCommit(false);
default autocommit is true
SQL statements are executed and commited as individual transaction
a transaction begins-->execute the statement successfully-->commit
rollback
any point before commiting a transaction
conn.rollback();
How to start
explicit
then set autocommit back to true
debug
DriverManager.setLogWriter(pw);
java.sql
elements
DriverManager
Connection
Statement
execute()
executeBatch()
executeQuery()
executeUpdate()
PreparedStatement
CallableStatement
ResultSet
get index start from 1
get table row count
SQLException
javax.sql
javax.sql.DataSource
javax.sql.ConnectionPoolDataSource
javax.sql.PooledConnection
javax.sql.XAConnection
javax.sql.XADataSource
metadata
DatabaseMetaData
ResultSetMetaData
ParameterMetaData
RowSetMetaData
Special
Datatype
Blob
Clob
Date/Time/Timestamp
talent sprint

PreparedStatement

Using `PreparedStatement` in place of `Statement` interface will improve the performance of a JDBC program. When a SQL statement is sent to database, the following tasks are performed

- The SQL statement's syntax should be verified to know whether it is correct or not. The SQL statement is divided into small pieces called `‘tokens’`. These tokens are also verified to know whether they are in `SQL` format or not.
- The another verification is done to know whether the table mentioned in the statement exists or not.

The above two statements are called `‘parsing’` and takes some time. When a SQL statement is executed, `Statement` interfcae does parsing every time the statement is executed and takes more time. On the other hand, if we use `PreparedStatement`, it does parsing only once and saves time. Hence `PreparedStatement` offers better performance.

We should create `PreparedStatement` object using `prepareStatement()` method of `Connection` interface as

```
PreparedStatement stmt = con.prepareStatement("INSERT
INTO Student VALUES(?, ?, ?, ?)");
```

Here, `‘?’` represents the value to be passed to the columns of the row. The first `?` represents the value to be passed to the first column and the second `?` represents the value for second column. These values can be passed using `setXXX()` method as

```
setInt (1, value);
setString (2, value);
```

Example Program to insert records using PreparedStatement.

```
import java.sql.*;
class PsInsertExample {
    public static void main(String a[]) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
            Connection con = DriverManager.getConnection("jdbc:mysql://
        localhost/db_test", "root", "root");
            String query = "INSERT into Student VALUES (?, ?, ?, ?)";
```

```
                PreparedStatement pstmt = con.prepareStatement(query);
                pstmt.setInt(1, 105);
                pstmt.setString(2, ''Prashanth'');
                pstmt.setInt(3, 24);
                pstmt.setString(4, ''Kerela'');
                pstmt.executeUpdate();
                pstmt.setInt(1, 106);
                pstmt.setString(2, ''Chandu'');
                pstmt.setInt(3, 21);
                pstmt.setString(4, ''Delhi'');
                pstmt.executeUpdate();
                System.out.println(''Rows Inserted Sucessfully'');
            }
            catch (SQLException e) {
                System.out.println(''Exception'' +e.getMessage());
            }
            catch (Exception e) {
                System.out.println(''Exception'' +e.getMessage());
            }
            finally {
                if (con != null) {
                    System.out.println(''Closing connection'');
                    con.close();
                } else {
                    System.out.println(''Connection Closed'');
                }
            }
        }
    }
```

Working With ResultSet Interface

`ResultSet` is an interface that manages the resulting data returned from a `Statement`.

ResultSet objects

- `ResultSet` maintains a cursor to the returned rows. The cursor is initially pointing before the first row.
- The `ResultSet.next()` method is called to position the cursor in the next row.
- The default `ResultSet` is not updatable and has a cursor that points only forward.
- It is possible to produce `ResultSet` objects that are scrollable and/or updatable as follows.

```
Statement stmt = con.createStatement
                    (ResultSet.TYPE_SCROLL_INSENSITIVE,
                     ResultSet.CONCUR_UPDATABLE);
```

ResultSet methods

boolean next() moves the cursor to the one row next from the current position.

boolean previous() moves the cursor to the one row previous from the current position.

boolean first() moves the cursor to the first row in result set object.

boolean last() moves the cursor to the last row in result set object.

boolean absolute(int row) moves the cursor to the specified row number in the ResultSet object.

boolean relative(int row) moves the cursor to the relative row number in the ResultSet object, it may be positive or negative.

int getInt(int columnIndex) returns the data of specified column index of the current row as int.

int getInt(String columnName) returns the data of specified column name of the current row as int.

String getString(int columnIndex) returns the data of specified column index of the current row as String.

String getString(String columnName) returns the data of specified column name of the current row as String.

Example Program to retrieve records.

```
import java.sql.*;
class SelectExample {
    public static void main(String a[]) {
        try {
            Class.forName(“com.mysql.jdbc.Driver”);
            Connection con = DriverManager.getConnection(“jdbc:mysql: //
    localhost/db_test ”, “root ”, “root ”);
            Statement stmt = con.createStatement();
            String query = “SELECT * FROM Student”;
            ResultSet rs = stmt.executeQuery(query);
            while (rs.next()) {
            System.out.println (“ current Row : ” +rs.getRow());
            System.out.println (“ Id : ” +rs.getInt (“ sid”));
            System.out.println (“ Name :”+rs.getString (“ sname”));
            System.out.println (“ Age : ” +rs.getInt(3));
        }
        catch (SQLException e) {
            System.out.println (“ Exception” +e.getMessage());
        }
        catch (Exception e) {
            System.out.println (“ Exception” +e.getMessage());
        }
        finally {
            if (con != null) {
                System.out.println (“ Closing connection”);
                con.close();
            } else {
                System.out.println (“ Connection Closed”);
            }
        }
    }
}
```

ResultSetMetaData Interface

`ResultSetMetaData` interface is used to describe a JDBC Driver object that encapsulates the metadata of the cursor. The `ResultSetMetaData` object is used to retrieve information about the columns in a `ResultSet`. It can also be used to get the information like column names, column datatypes, and column length, related to particular result. The `ResultSetMetaData` object is obtained by using the `ResultSet` object, that is by invoking the `getMetaData()` method on the `ResultSet` object.

Methods of ResultSetMetaData

getColumnCount() returns an integer value and retrieves the number of columns in the `ResultSet` object.

getColumnName(int index) takes the column index and returns the column name.

getColumnType(int index) takes the column index and returns the integer value that indicates the columns SQL type.

getColumnLabel(int index) retrieves the title of the specified column in the `ResultSet` object.

getTableName(int column) retrieves the name of the specified column in the `ResultSet` object.

getColumnTypeName(int index) takes the column index and returns the database specific type name.

Example Program to demonstrate ResultSetMetaData.

```
import java.sql.*;
class RSMetaDataExample {
    public static void main(String a[]) {
        try {
            Class.forName(''com.mysql.jdbc.Driver'');
            Connection con = DriverManager.getConnection(''jdbc:mysql://localhost/db_test'', ''root'', ''root'');
            Statement stmt = con.createStatement();
            String query = ''SELECT * FROM Student '';
            ResultSet rs = stmt.executeQuery(query);
```

```
                ResultSetMetaData md = rs.getMetaData();
                System.out. println (" Column count : " +md.getColumnCount());
                System.out. println (" second column Name : " +md.getColumnName
        (2));
                System.out. println (" Column Data Type : " +md.
        getColumnTypeName(2));
            }
            catch (SQLException e) {
                System.out. println (" Exception" +e.getMessage());
            }
            catch (Exception e) {
                System.out. println (" Exception" +e.getMessage());
            }
            finally {
                if (con != null) {
                    System.out. println (" Closing connection");
                    con. close ();
                } else {
                    System.out. println (" Connection Closed");
                }
            }
        }
}
```

Working With Batch Updates

The batch update option allows you to submit multiple Data Manipulation Language (DML) operations to a database as a single unit. Submitting multiple DML statements together, rather than submitting them individually, improves the performance of the system by reducing the time consumption in executing statements.

Batch Update Methods

addBatch(String SQLStatement) adds the given SQL statement to the current batch i.e, the batch that is managed by this `Statement` object.

clearBatch() clears the batch currently assosiated with this Statement object.

int[]executeBatch() submits the current batch on this `Statement` object to the underlying datasource. Once the batch is submitted to the data source, the statements in a batch are executed in the sequence in which thay have been added to the batch. The following are the outcomes of this method.

- If the batch executes successfully, this method returns an array of integer values whose length is equal to the number of statements in the batch.
- If the value of any element in this array is equal to STATEMENT.SUCCESS_NO_INFO, it indicates that the statement has been executed successfully but the number of rows affected is unknown.
- In case a statement in a batch fails to be executed or produces a result set , further processing of the batch execution depends on the database that is in use. The database may continue to execute the batch or may terminate it.

Example Program to demonstrate Batch Updates.

```
import java.sql.*;
class BatchExample {
    public static void main(String args[]) {
        Connection con = null;
        try {
            Class.forName(''com.mysql.jdbc.Driver'');
            con = DriverManager.getConnection(''jdbc:mysql://localhost/db_test
    '', ''root'', ''root'');
            Statement stmt = con.createStatement();
            String query1 = ''INSERT INTO EMP VALUES (109, 'albert', 9000)''
    ;
            String query2 = ''INSERT INTO EMP VALUES (110, 'robert', 8000)''
    ;
            String query3 = ''UPDATE EMP SET sal = sal + 1000'';
            String query4 = ''DELETE FROM EMP WHERE empno = 109'';
            stmt.addBatch(query1);
            stmt.addBatch(query2);
            stmt.addBatch(query3);
            stmt.addBatch(query4);
            int a[] = stmt.executeBatch();
            System.out.println(a[0] + '' row(s) inserted'');
```

```
                System.out.println(a[1] + " row(s) inserted");
                System.out.println(a[2] + " row(s) updated");
                System.out.println(a[3] + " row(s) deleted");
            } catch(Exception e) {
                System.out.println("Exception: " + e.getMessage());
            }
            finally {
                try {
                    con.close();
                } catch(Exception e){}
            }
        }
    }
```

DataBaseMetaData Interface

`DataBaseMetaData` is an interface implemented by the JDBC driver provider. The `DataBaseMetaData` object encapsulates all the information related to the database and driver. The `DataBaseMetaData` object is useful to get the database details, such as DB vendor name, product name, version etc. Use the `Connection.getMetaData()` method to get the `DataBaseMetaData` object.

Methods of DataBaseMetaData

int getDataBaseMajorVersion() retrieves the major version number of the database in use.

int getDataBaseMinorVersion() retrieves the minor version number of the database in use.

String getDataBaseProductName() retrieves the name of this database product.

String getDataBaseProductVersion() retrieves the version number of this database product.

int getDriverMajorVersion() retrieves the major version number of the JDBC Driver.

int getDriverMinorVersion() retrieves the minor version number of the JDBC Driver.

int getMaxRowSize() retrieves the maximum number of bytes a database allows in a single row.

int getMaxConnections() retreives the maximun number of concurrent connections to a database that are possible.

String getDriverName() retrieves the name of this JDBC driver.

String getURL() retrieves the URL for this DBMS.

String getUserName() retrieves the user name as known to this database.

Example Program to demonstrate DataBaseMetaData.

```
import java.sql.*;
class DBMetaDataExample {
    public static void main(String a[]) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
            Connection con = DriverManager.getConnection("jdbc:mysql://localhost/db_test", "root", "root");
            DatabaseMetaData md = con.getMetaData();
            System.out.println("Product Name : "+md.getDatabaseProductName());
            System.out.println("Version : "+md.getDatabaseProductVersion());
            System.out.println("Driver Name : "+md.getDriverName());
            System.out.println("Major Version : "+md.getDatabaseMajorVersion());
            System.out.println("Minor Version : "+md.getDatabaseMinorVersion());
            System.out.println("Size:"+md.getMaxRowSize());
            System.out.println("URL : "+md.getURL());
            System.out.println("User : "+md.getUserName());
        }
        catch (SQLException e) {
            System.out.println("Exception" +e.getMessage());
        }
        catch (Exception e) {
            System.out.println("Exception" +e.getMessage());
        }
        finally {
            if (con != null) {
                System.out.println("Closing connection");
                con.close();
            } else {
                System.out.println("Connection Closed");
            }
        }
    }
}
```

Transactions

`Transaction` is a set of statements executed on a resource or resources applying ACID properties. In multiple environments various users work on databases and there may be chances of data loss. Therefore, to avoid data loss, DBMS is used to maintain data integrity within database. `Transactions` are required to provide data integrity, correct application semantics, and a consistent view of data during concurent acess. In general, DBMS provides `atomicity`, `consistency`, `isolation`, and `durability` for each transaction in a database. These properties are collectively called as the `ACID` properties.

Describing ACID properties

Atomicity is the ability of a transaction to hold the decision of saving the statement until the end of the set of statements i.e, transactions.

Consistency guarantees that the data remains in a legal state when the transaction begins and ends.

Isolation is the ability of the transaction to isolate or hide the data from other transactions until it ends.

Durability guarantees that once the user has been notified of the successful transaction i.e, if a transaction is commited , it will persist all the statements or it will leave complete transaction unsaved.

JDBC Transactions

By default, when a Connection is created, transaction is in auto-commit mode.

- Each individual SQL statement is treated as transaction and automatically commited after it is executed.
- To group two or more statements together, disable auto-commit mode.

  ```
  Connection.setAutoCommit(false);
  ```

- Explicitly call the commit method to complete the transaction with in the database.

  ```
  Connection.commit();
  ```

- Transactions can be rollbacked programatically in the event of failure.

  ```
  Connection. rollback ();
  ```

By default, JDBC auto commits all SQL statements. However when you want to create an atomic operations that involves multiple SQL statements, disable auto-commit. After auto-commit is disabled no SQL statements are commited to the database until commit method is called explicitly. The other advantage of managing transactions is the ability to rollback the set of statements in the event of failure using rollback method.

Note
Create table and Insert the data as given below in the database.
CREATE TABLE Bank(accno int, bal int);
INSERT INTO Bank VALUES(101, 10000);
INSERT INTO Bank VALUES(102, 10000);

Example Program to demonstrate Transactions.

```
import java.sql.*;
class TransactionExample {
    public static void main(String a[]) {
        Connection con = null;
        try {
            Class.forName("com.mysql.jdbc.Driver");
            con = DriverManager.getConnection("jdbc:mysql://localhost:3306/
    db_test", "root", "root");
```

```
                Statement stmt = con.createStatement();
                con.setAutoCommit(false);
                stmt.addBatch("update Bank set bal = bal - 1000 where accno =
        7900");
                stmt.addBatch("update Bank set bal = bal + 1000 where accno =
        000"); // fails updating record
                int a[] = stmt.executeBatch();
                for (int i = 0; i < 2; i++) {
                    if (a[i] != 0) {
                        System.out.println("Transaction is success..");
                        return;
                    }
                }
                System.out.println("Transaction is fail..");
                con.rollback();
            } catch (Exception e) {
                System.out.println("Exception" + e.getMessage());
                System.out.println("Transaction is failed..");
                con.rollback();
            }
            finally {
                try {
                    con.commit();
                    con.close();
                } catch(Exception e){}
            }

        }
    }
```