**1.Preparecall():-** You can create an object of the CallableStatement (interface) using the prepareCall() method of the Connection interface. This method accepts a string variable representing a query to call the stored procedure and returns a CallableStatement object.

**Syntax for preparecall():-**

public java.sql.CallableStatement prepareCall(java.lang.String sql)

**2.resultsetmetadata:-**

ResultSetMetaData is an interface in java.sql package of JDBC API which is used to get the metadata about a ResultSet object. Whenever you query the database using SELECT statement, the result will be stored in a ResultSet object. Every ResultSet object is associated with one ResultSetMetaData object. This object will have all the meta data about a ResultSet object like schema name, table name, number of columns, column name, datatype of a column etc. You can get this ResultSetMetaData object using getMetaData() method of ResultSet.

getMetaData() method of java.sql.ResultSet interface returns ResultSetMetaData object associated with a ResultSet object. Below is the syntax to get the ResultSetMetaData object.

**Syntax:- ResultSetMetaData rsmd = rs.getMetaData();**

**3.Driver Types:-**

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

1. JDBC-ODBC bridge driver
2. Native-API driver (partially java driver)
3. Network Protocol driver (fully java driver)
4. Thin driver (fully java driver)

**JDBC-ODBC bridge driver:**

|  |
| --- |
| The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver. |

### Native-API driver:

|  |
| --- |
| The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. |

**Network Protocol driver:**The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol.

4) Thin driver

|  |
| --- |
| The thin driver converts JDBC calls directly into the vendor-specific database protocol  So known as thin driver. |

4.**SQL Exception:-**

In JDBC, we may get exceptions when we execute or create the query. Exceptions that occur due to the Database or Driver come under SQL Exception. Using Exception handling, we can handle the SQL Exception like we handle the normal exception.

SQLException is available in the java.sql package. It extends the Exception class which means that we can use the methods available in the Exception class in the SQLException class as well.

Example for SQL Exception

Syntax error in the SQL statement may result in SQL Exception. When such an exception occurs, an object of the SQLException class will be passed to the catch block. By using the information in the SQLException object, we can catch that exception and continue the program.

There are three types of catogeries:

* SQLNonTransientException.
* SQLTransientException
* SQLRecoverableException

**To handle the sql exception**:

JDBC-related exception mostly throws SQLException, and it is a checked exception so we must either catch it or throw it. All the business logic and commit data should be done in a Try block, if any exception happened in the block we should catch and handle it in the Catch block. Based on the exception type, we should do the rollbacks or commit in the Catch block.

**5.Execute and ExecuteQuery:-**

* **execute :**
* This method is used to execute SQL DDL statements, it returns a boolean value specifying weather the ResultSet object can be retrieved
* Returns true if the first object that the query returns is a ResultSet object. ...
* **executeQuery** :
* Returns one ResultSet object.

Execute Query is an operating system independent database utility written entirely in Java. Using the flexibility provided by Java Database Connectivity (JDBC), Execute Query provides a simple way to interact with almost any database from simple queries to table creation and import/export of an entire schema's data.

**6.How to perform transactions(commit and rollback) in jdbc?**

**Rollback:-**

A rollback operation **undoes all the changes done by the current transaction** i.e. If you call a rollBack() method of the Connection interface, all the modifications are reverted until the last commit.Con.rollback()

When we are making changes in the database through a java.sql.Connection, it’s necessary prevent it form going to an inconsistent state, There are some key steps.

* Manage manually the changes with transactions, disabling the auto commit
* Explicitly call commit and rollback
* Rollback a transaction in a catch clause
* Explicitly close the connection.

**Example**:

package com.java;

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.SQLException;

public class Rollback {

private static String INSERT = "INSERT INTO test.department (idDepartment, name) VALUES (?, ?)";

    public static void insertRow(Connection conn, int idRow, String contentRow)

            throws SQLException {

        PreparedStatement pstmt = null;

        pstmt = conn.prepareStatement(INSERT);

        pstmt.setInt(1, idRow);

        pstmt.setString(2, contentRow);

        pstmt.execute();

        pstmt.close();

    }

    public static void main(String[] args) {

        Connection connection = null;

        try {

            // 1st Step, Make a connection

            connection = DBConnection.getConnection();

        } catch (SQLException e) {

            System.err.println("There was an error getting the connection");

        }

        try {

            // 2nd Step, Disable the auto commit

            connection.setAutoCommit(false);

            System.err.println("The autocommit was disabled!");

        } catch (SQLException e) {

            System.err.println("There was an error disabling autocommit");

        }

        // Starts JDBC Transaction

        try {

            // 3rd Step, Execute the statements

            insertRow(connection, 1, "Malaga");

            insertRow(connection, 2, "Barcelona");

            // 4th Step, Complete a transaction, committing the changes.

            connection.commit();

            System.err.println("The transaction was successfully executed");

        } catch (SQLException e) {

            try {

                // 5th and Final Step, We must rollback the transaction if a

                // SQLException occurs

                connection.rollback();

                System.err.println(e.getMessage());

                System.err.println("Transaction rollback");

            } catch (SQLException e1) {

                System.err.println(e1.getMessage());

                System.err.println("There was an error making a rollback");

            }

        }

    }

}

**Commit**:

Commit command is used to permanently save any transaction into the database. It is used to end your current transaction and make permanent all changes performed in the transaction. A transaction is a sequence of SQL statements that Oracle Database treats as a single unit. This statement also erases all save points in the [transaction](https://www.google.com/url?client=internal-element-cse&cx=009682134359037907028:tj6eafkv_be&q=https://www.geeksforgeeks.org/sql-transactions/&sa=U&ved=2ahUKEwjg99Sv4ZDtAhVVmuYKHSMEASgQFjAAegQIAhAC&usg=AOvVaw1oDg630otyQkShj46G_LUW) and releases transaction locks.

When we use any [DML](https://www.google.com/url?client=internal-element-cse&cx=009682134359037907028:tj6eafkv_be&q=https://www.geeksforgeeks.org/dml-full-form/&sa=U&ved=2ahUKEwj5q_LU4JDtAhXe7XMBHTvFBHgQFjACegQICBAC&usg=AOvVaw2T2iVgnxpTGd_MiUEO_Wml) command like INSERT, UPDATE or DELETE, the changes made by these commands are not permanent, until the current session is closed, the changes made by these commands can be rolled back. To avoid that, we use the COMMIT command to mark the changes as permanent.

**Syntax of Commit**: Con.commit

**Example:**

public void updateCoffeeSales(HashMap<String, Integer> salesForWeek) throws SQLException {

String updateString =

"update COFFEES set SALES = ? where COF\_NAME = ?";

String updateStatement =

"update COFFEES set TOTAL = TOTAL + ? where COF\_NAME = ?";

try (PreparedStatement updateSales = con.prepareStatement(updateString);

PreparedStatement updateTotal = con.prepareStatement(updateStatement))

{

con.setAutoCommit(false);

for (Map.Entry<String, Integer> e : salesForWeek.entrySet()) {

updateSales.setInt(1, e.getValue().intValue());

updateSales.setString(2, e.getKey());

updateSales.executeUpdate();

updateTotal.setInt(1, e.getValue().intValue());

updateTotal.setString(2, e.getKey());

updateTotal.executeUpdate();

con.commit();

}

} catch (SQLException e) {

JDBCTutorialUtilities.printSQLException(e);

if (con != null) {

try {

System.err.print("Transaction is being rolled back");

con.rollback();

} catch (SQLException excep) {

JDBCTutorialUtilities.printSQLException(excep);

}

}

}

}

**7.ACID Properties**:-

ACID is a concept (and an acronym) that refers to the four properties of a transaction in a database system,

which are: Atomicity, Consistency, Isolation and Durability.

**Atomicity**   
By this, we mean that either the entire transaction takes place at once or doesn’t happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.   
**Abort**: If a transaction aborts, changes made to database are not visible.   
**Commit**: If a transaction commits, changes made are visible.   
Atomicity is also known as the ‘All or nothing rule’.

**Consistency**:  
This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database.

**Isolation** :  
This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed. This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order.

**Durability:**This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs. These updates now become permanent and are stored in non-volatile memory. The effects of the transaction, thus, are never lost.

The ACID properties, in totality, provide a mechanism to ensure correctness and consistency of a database in a way such that each transaction is a group of operations that acts a single unit, produces consistent results, acts in isolation from other operations and updates that it makes are durably stored.