# Java Database Connectivity (JDBC)

While developing an IT application you will come across scenarios where you will have to store / persist data available in a program (Object states) for a longer duration, so that data can be retrieved at a later point of time if needed for processing. This is where JDBC comes handy. It helps java program to persist data into Database. JDBC is a Java API which enables Java Programs to connect to Databases, to store, search, modify and delete data. JDBC API is an interface based, industry standard for connectivity between Java programs and SQL based Databases. The JDBC standards are defined using Java Interfaces defined in java.sql package.

**JDBC Driver**

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)

**1) 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.

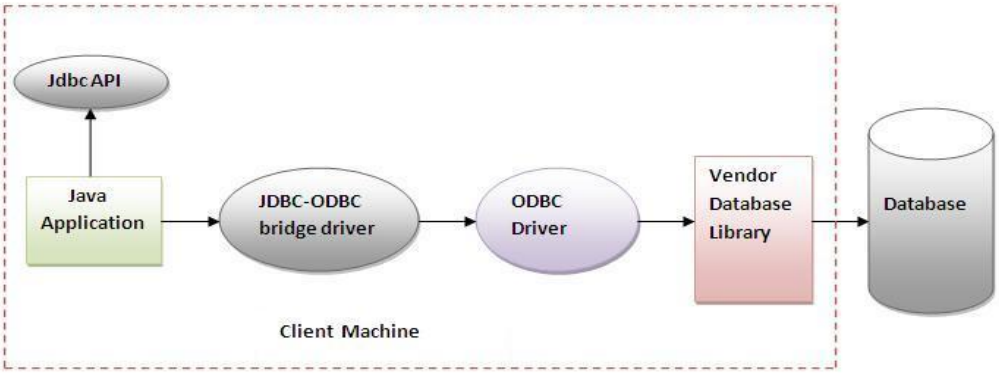



Figure- JDBC-ODBC Bridge Driver

**Advantages:**

- ✓ Easy to use.
- ✓ Can be easily connected to any database.

**Disadvantages:**

- ✓ Performance degraded because JDBC method call is converted into the ODBC function calls.
- ✓ The ODBC driver needs to be installed on the client machine.

**2) 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. It is not written entirely in java.

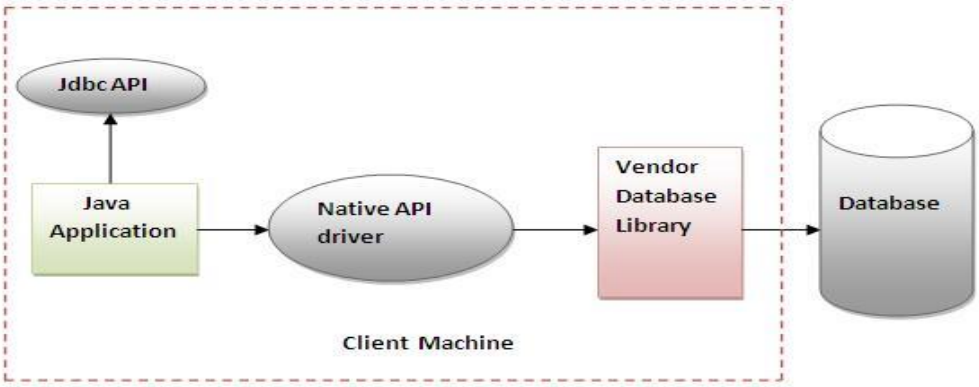



Figure- Native API Driver

**Advantage:**

- ✓ Performance upgraded than JDBC-ODBC bridge driver.

**Disadvantage:**

- ✓ The Native driver needs to be installed on the each client machine.
- ✓ The Vendor client library needs to be installed on client machine.

**3) Network Protocol driver**

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.

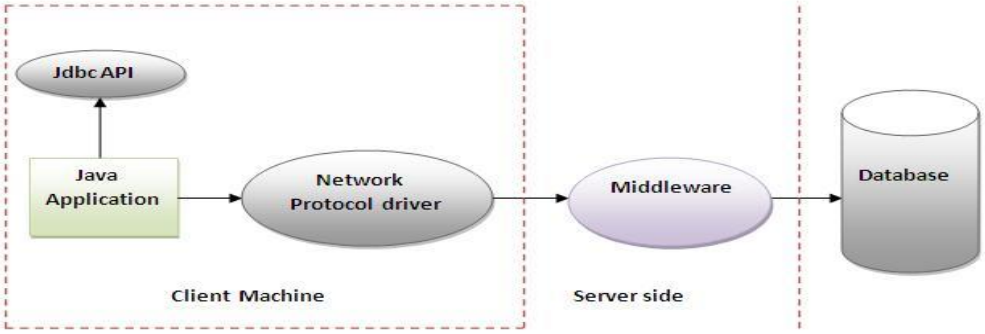



Figure- Network Protocol Driver

**Advantage:**

- ✓ No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

**Disadvantages:**

- ✓ Network support is required on client machine.
- ✓ Requires database-specific coding to be done in the middle tier.
- ✓ Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

**4) Thin driver**

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.

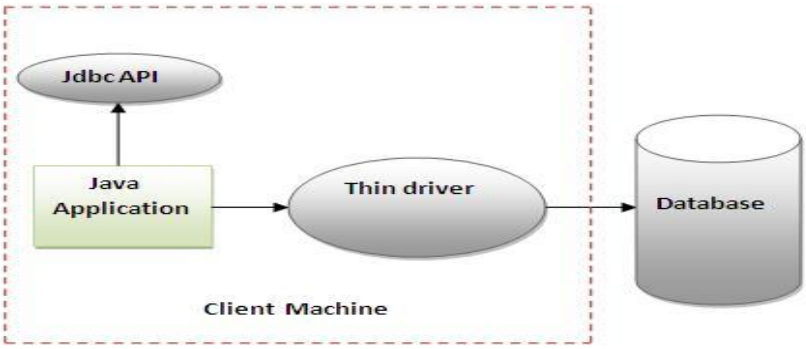



Figure- Thin Driver

**Advantage:**

- ✓ Better performance than all other drivers.
- ✓ No software is required at client side or server side.

**Disadvantage:**

- ✓ Drivers depends on the Database.

## JDBC Connection Steps

There are seven standard steps in querying databases

1) Load the JDBC driver.
2) Prepare connection URL.
3) Establish the connection.
4) Create a statement object.
5) Execute a query.
6) Process the results.
7) Close the connection.

### 1) Load the JDBC Driver :

Load the driver class by calling **Class.forName( )** with the Driver class name as an argument. Once loaded, the Driver class creates an instance of itself. A client can connect to Database Server through JDBC Driver. Since most of the Database servers support ODBC driver therefore JDBC-ODBC Bridge driver is commonly used.

The return type of the Class.forName(String ClassName) method is "Class". Class is a class in java.lang package.

**Example:**

```
try {
                Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
                //Class.forName("connect.microsoft.MicrosoftDriver");   for MS driver
                //Class.forName("oracle.jdbc.driver.OracleDriver");      for Oracle driver
                //Class.forName("com.mysql.jdbc.driver" );               for MySQL driver
    }
catch(ClassNotFoundException e) {
        System.out.println("Error loading driver: " + e);
}
```

The string taken in the Class.forName() method is the fully qualified class name. Anyone can check vendor's documentation to find it out. Most database vendors supply free JDBC drivers for their databases.

### 2) Prepare Connection URL:

Once you have loaded the JDBC driver, you need to specify the location of the database server. Generally, the url format is: **jdbc:vendordb:userinfo** plus server host, port number and database name. You should check vendors documentation for exact information about such format. The following list two examples:

```
//for one of Oracle drivers
    String host       = "dbhost.yourcompany.com";
    String dbName = "someName";
        int port      = 1234;
 String oracleURL = "jdbc:oracle:thin:@" + host +  ":" + port + ":" + dbName;
```

**3) Establish the Connection :**

To make the actual network connection, pass the URL, the database username, and the password to the getConnection method of the DriverManager class, as illustrated in the following example.

```
String username = "scott";
String password = "tiger";
Connection connection = DriverManager.getConnection(oracleURL, username, password);
```

**4) Create a Statement Object:**

A Statement object is used to send queries and commands to the database and is created from the Connection as follows:

```
Statement statement = connection.createStatement();
```

**5) Execute a Query :**

Once you have a Statement object, you can use it to send SQL queries by using the executeQuery method, which returns an object of type ResultSet. Here is an example:

```
String query = "SELECT * FROM emp";
ResultSet resultSet = statement.executeQuery(query);
```

- ✓ **public ResultSet executeQuery(String sql):** is used to execute SELECT query. It returns the object of ResultSet.
- ✓ **public int executeUpdate(String sql):** is used to execute specified query, it may be create, drop, insert, update, delete etc.
- ✓ **public boolean execute(String sql):** is used to execute queries that may return multiple results.
- ✓ **public int[] executeBatch():** is used to execute batch of commands.

**6) Process the Result:**

The object of ResultSet maintains a cursor pointing to a particular row of data. Initially, cursor points to before the first row. The simplest way to handle the results is to process them one row at a time, using the ResultSet's next method to move through the table a row at a time.

Within a row, ResultSet provides various getXXX() methods that take a column index or column name as an argument and return the result as a variety of different Java types. For instance, use getInt if the value should be an integer, getString for a String, and

so on for most other data types. If you just want to display the results, you can use getString regardless of the actual column type. However, if you use the version that takes a column index, note that columns are indexed starting at 1, not at 0 as with arrays, vectors, and most other data structures in the Java programming language.

| Method | Description |
|---|---|
| ResultSet.first() | Moves the cursor to the first row in this ResultSet object. |
| ResultSet.last() | Moves the cursor to the last row in this ResultSet object. |
| ResultSet.next() | Moves the cursor forward one row from its current position. |
| ResultSet.previous() | Moves the cursor to the previous row in this ResultSet object. |
| ResultSet.getBoolean() | Returns the value as boolean of a given column. |
| ResultSet.getByte() | Returns the value as byte of a given column. |
| ResultSet.getDate() | Returns the value as Date of a given column. |
| ResultSet.getDouble() | Returns the value as double of a given column. |
| ResultSet.getFloat() | Returns the value as float of a given column. |
| ResultSet.getInt() | Returns the value as int of a given column. |
| ResultSet.getLong() | Returns the value as long of a given column. |
| ResultSet.getString() | Returns the value as String of a given column. |
| ResultSet.close() | Closes the resources used by this ResultSet object. |

```
while(resultSet.next())
  {
        System.out.println(    results.getString(1)    + " \t  "
                             + results.getString(2)    + " \t  "
                             + results.getString(3)  );
  }
```

7) **Close the Connection :** To close the connection explicitly, we should do:

```
connection.close();
```

We should postpone this step if we expect to perform additional database operations, since the overhead of opening a connection is usually large. In fact, reusing existing connections is such an important optimization.

**Example:** // Display Employee details

```
import java.sql.*;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
public class Login
{
    public static void main(String args[]) throws Exception
    {
        Connection con = null;
        Statement stmt  = null;
      try{
        Class.forName("oracle.jdbc.driver.OraceDriver");
        con=DriverManager.getConnection
                ("jdbc:oracle:thin:@localhost:1521:XE","scott","tiger");
        stmt=con.createStatement();
        ResultSet rs=stmt.executeQuery("select * from emp");
            while(rs.next())
            {
                System.out.println(rs.getString(1) +"\t" +rs.getString(2));
            }
         }
         catch(ClassNotFoundException e) {
            System.out.println("Message: " +e);
         }
        catch(SQLException se) {
             System.out.println("Error: " +se.getMessage());
         }
        finally {
            stmt.close();
            con.close();
          }
     }
}
```

## Types of JDBC Statements:

Once a connection is obtained we can interact with the database. The JDBC Statement, CallableStatement and PreparedStatement interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database. They also define methods that help bridge data type differences between Java and SQL data types used in a database.

Below Table provides a summary of each interface purpose to understand how do you decide which interface to use?

| Interfaces | Recommended Use |
|---|---|
| **Statement** | Use for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters. |
| **PreparedStatement** | Use when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime. |
| **CallableStatement** | Use when you want to access database stored procedures. The CallableStatement interface can also accept runtime input parameters. |

## Creating Statement Object

Before you can use a Statement object to execute a SQL statement, you need to create one using the Connection object's createStatement( ) method, as in the following example:

```
Statement stmt = null;
try {
  stmt = conn.createStatement( );
  . . .
}
catch (SQLException e) {
  . . .
}
finally {
  . . .
}
```

Once you've created a Statement object, you can then use it to execute a SQL statement with one of its three execute methods.

- **ResultSet executeQuery(String SQL) :** Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.

- **int executeUpdate(String SQL) :** Returns the numbers of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.

- **boolean execute(String SQL) :** Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.

**Closing Statement Obeject:**

Just as you close a Connection object to save database resources, for the same reason you should also close the Statement object.

A simple call to the close() method will do the job. If you close the Connection object first it will close the Statement object as well. However, you should always explicitly close the Statement object to ensure proper cleanup.

```
Statement stmt = null;
try {
  stmt = conn.createStatement( );
  . . .
}
catch (SQLException e) {
  . . .
}
finally {
  stmt.close();
}
```

**Example:**

```
// Display Student name and Roll Number using Statement Object
import java.sql.*;

public class DBApplication{

    public static void main(String args[])
    {

    try{
        Class.forName("oracle.jdbc.driver.OracleDriver");
```

```
Connection con =
      DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","system","abc123");

Statement stmt = con.createStatement();

            String sqlQuery ="select *from student";

            ResultSet rs= stmt.executeQuery(sqlQuery);

            while(rs.next())
            {
                        System.out.println(rs.getString(1) +"\t " +rs.getString(2) );
             }
            con.close();
            }
        catch(ClassNotFoundException cnf)
        {
            System.out.println("Message: " +cnf);
        }
        catch(SQLException se)
        {
            System.out.println("Error: " +se);
        }

    }
}
```

## The PreparedStatement Objects

The PreparedStatement interface extends the Statement interface which gives you added functionality with a couple of advantages over a generic Statement object. This statement gives you the flexibility of supplying arguments dynamically. Creating PreparedStatement Object:

```
PreparedStatement pstmt = null;
try {
   String SQL = "Update Employees SET age = ? WHERE id = ?";
   pstmt = conn.prepareStatement(SQL);
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   . . .
}
```

All parameters in JDBC are represented by the ? symbol, which is known as the parameter marker. You must supply values for every parameter before executing the SQL statement.

The setXXX() methods bind values to the parameters, where XXX represents the Java data type of the value you wish to bind to the input parameter. If you forget to supply the values, you will receive an SQLException.

Each parameter marker is referred to by its ordinal position. The first marker represents position 1, the next position 2, and so forth. This method differs from that of Java array indices, which start at 0.

All of the Statement object's methods for interacting with the database (a) execute(), (b) executeQuery(), and (c) executeUpdate() also work with the PreparedStatement object. However, the methods are modified to use SQL statements that can take input the parameters.
Closing PreparedStatement Obeject:

Just as you close a Statement object, for the same reason you should also close the PreparedStatement object.

A simple call to the close() method will do the job. If you close the Connection object first it will close the PreparedStatement object as well. However, you should always explicitly close the PreparedStatement object to ensure proper cleanup.

```
PreparedStatement pstmt = null;
try {
   String SQL = "Update Employees SET age = ? WHERE id = ?";
   pstmt = conn.prepareStatement(SQL);
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   pstmt.close();
}
```

**Example:**

```
// Registration Form Using PreparedStatement Object
import java.util.*;
import java.sql.*;

public class DBP
  {
 public static void main(String args[]) throws Exception
      {
```

```
 try{
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter your Name:");
        String Nam = sc.next();
                System.out.print("Enter User id:");
                String uid = sc.next();
        System.out.print("Enter Password:");
        String passKey = sc.next();
                System.out.print("Mobile No:");
                String mobile = sc.next();
        System.out.print("Enter your email id:");
        String email = sc.next();

Class.forName("oracle.jdbc.driver.OracleDriver");

Connection con=
DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","system","abc123");

PreparedStatement ps=con.prepareStatement("insert into registeruser values(?,?,?,?,?)");

                ps.setString(1,Nam);
                ps.setString(2,uid);
                ps.setString(3,passKey);
                ps.setString(4,mobile);
                ps.setString(5,email);

                int i=ps.executeUpdate();
        if(i>0)
            System.out.print("You are successfully registered...");
        else
            System.out.print("You are Not registered...");
}
catch (Exception e2) {
        System.out.println(e2);
    }
  }
```

## The CallableStatement Objects

Like Connection object creates the Statement and PreparedStatement objects, it also creates the CallableStatement object which would be used to execute a call to a database stored procedure.

### Creating CallableStatement Object:

Suppose, you need to execute the following Oracle stored procedure:

```
CREATE OR REPLACE PROCEDURE getEmpName
                    (EMP_ID IN NUMBER, EMP_FIRST OUT VARCHAR) AS

BEGIN
  SELECT first INTO EMP_FIRST  FROM Employees WHERE ID = EMP_ID;
END;
```

**NOTE:** Above stored procedure has been written for Oracle, but we are working with MySQL database so let us write same stored procedure for MySQL as follows to create it in EMP database:

Three types of parameters exist: IN, OUT, and INOUT. Here are the definitions of each:

| Parameter | Description |
|---|---|
| IN | A parameter whose value is unknown when the SQL statement is created. You bind values to IN parameters with the setXXX() methods. |
| OUT | A parameter whose value is supplied by the SQL statement it returns. You retrieve values from theOUT parameters with the getXXX() methods. |
| INOUT | A parameter that provides both input and output values. You bind variables with the setXXX() methods and retrieve values with the getXXX() methods. |

The following code snippet shows how to employ the Connection.prepareCall() method to instantiate a CallableStatement object based on the preceding stored procedure:

```
CallableStatement cstmt = null;
try {
  String SQL = "{call getEmpName (?, ?)}";
  cstmt = conn.prepareCall (SQL);
  . . .
}
catch (SQLException e) {
  . . .
}
finally {
  . . .
}
```

Using CallableStatement objects is much like using PreparedStatement objects. You must bind values to all parameters before executing the statement, or you will receive an SQLException. If you have IN parameters, just follow the same rules and techniques that apply to a PreparedStatement object; use the setXXX() method that corresponds to the Java data type you are binding.

When you use OUT and INOUT parameters you must employ an additional CallableStatement method, registerOutParameter(). The registerOutParameter() method binds the JDBC data type to the data type the stored procedure is expected to return.

Once you call your stored procedure, you retrieve the value from the OUT parameter with the appropriate getXXX() method. This method casts the retrieved value of SQL type to a Java data type.

### Closing CallableStatement Obeject

Just as you close other Statement object, for the same reason you should also close the CallableStatement object. A simple call to the close() method will do the job. If you close the Connection object first it will close the CallableStatement object as well. However, you should always explicitly close the CallableStatement object to ensure proper cleanup.

```
CallableStatement cstmt = null;
try {
   String SQL = "{call getEmpName (?, ?)}";
   cstmt = conn.prepareCall (SQL);
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   cstmt.close();
}
```

**Example:**

```
import java.sql.*;

public class JDBCExample {
   public static void main(String[] args) {
         Connection conn = null;
         CallableStatement stmt = null;

try{
     Class.forName("oracle.jdbc.driver.OracleDriver");
    Connection                                                                    con=
DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","system","abc123");

     String sql = "{call getEmpName (?, ?)}";
     stmt = conn.prepareCall(sql);
```

```
    int empID = 102;
    stmt.setInt(1, empID);                    // This would set ID as 102

    stmt.registerOutParameter(2, java.sql.Types.VARCHAR);
    stmt.execute();

    String empName = stmt.getString(2);
    System.out.println("Emp Name with ID:" + empID + " is " + empName);
    stmt.close();
    conn.close();
  }catch(SQLException se){
    se.printStackTrace();
  }catch(Exception e){
    e.printStackTrace();
  }
 }
}
```

**Output:**

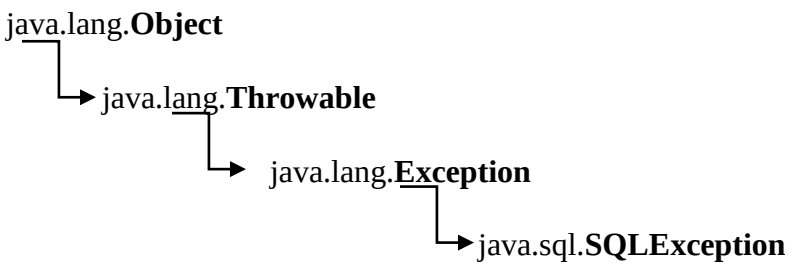
Emp Name with ID: 102 is  PHANI KUMAR

## SQLException

**Exception:** An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.

When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called throwing an exception

**SQLException:**




When JDBC encounters an error during an interaction with a data source, it throws an instance of SQLException as opposed to Exception.

**SQLException Constructors:**

**a) public SQLException()**

Constructs a SQLException object. The reason, SQLState are initialized to null and the vendor code is initialized to 0. The cause is not initialized, and may subsequently be initialized by a call to the Throwable.initCause(java.lang.Throwable) method.

**b) public SQLException(String reason, String SQLState)**

Constructs a SQLException object with a given reason and SQLState. The cause is not initialized, and may subsequently be initialized by a call to the Throwable.initCause(java.lang.Throwable) method. The vendor code is initialized to 0.

**Parameters:**

**reason** - a description of the exception
**SQLState** - an XOPEN or SQL:2003 code identifying the exception

**c) public SQLException(String reason)**

Constructs a SQLException object with a given reason. The SQLState is initialized to null and the vender code is initialized to 0. The cause is not initialized, and may subsequently be initialized by a call to the hrowable.initCause(java.lang.Throwable) method.

**Parameters:**

**reason** - a description of the exception

**d) public SQLException(String reason, String SQLState, int vendorCode)**

Constructs a SQLException object with a given reason, SQLState and vendorCode. The cause is not initialized, and may subsequently be initialized by a call to the Throwable.initCause(java.lang.Throwable) method.

**Parameters:**

**reason** - a description of the exception
**SQLState** - an XOPEN or SQL:2003 code identifying the exception
**vendorCode** - a database vendor-specific exception code

**Methods in SQLException Class:**

| Method | Description |
|---|---|
| int **getErrorCode()** | Retrieves the vendor-specific exception code for this SQLException object. |
| SQLException **getNextException()** | Retrieves the exception chained to this SQLException object by setNextException(SQLException ex). |
| String **getSQLState()** | Retrieves the SQLState for this SQLException object. |
| void **setNextException(SQLException ex)** | Adds an SQLException object to the end of the chain. |
| String **getMessage()** | Returns the detail message string of this throwable. |
| void **printStackTrace( )** | Prints the current exception, or throwable, and it's backtrace to a standard error stream. |
| void **printStackTrace(PrintStream s)** | Prints this throwable and its backtrace to the print stream you specify. |

## Java DatabaseMetaData interface

DatabaseMetaData interface provides methods to get meta data of a database such as database product name, database product version, driver name, name of total number of tables, name of total number of views etc. Commonly used methods of DatabaseMetaData interface

- **public String getDriverName() throws SQLException** - it returns the name of the JDBC driver.
- **public String getDriverVersion()throws SQLException** - it returns the version number of the JDBC driver.
- **public String getUserName()throws SQLException** - it returns the username of the database.
- **public String getDatabaseProductName()throws SQLException** - it returns the product name of the database.
- **public String getDatabaseProductVersion()throws SQLException** - it returns the product version of the database.
- **public ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types) throws SQLException** – it returns the description of the tables of the specified catalog. The table type can be TABLE, VIEW, ALIAS, SYSTEM TABLE, SYNONYM etc.

The getMetaData() method of Connection interface returns the object of DatabaseMetaData.

**Syntax:** public DatabaseMetaData getMetaData()throws SQLException

**//Simple Example of DatabaseMetaData interface**

```
import java.sql.*;

public class DBMetadata{
        public static void main(String args[]){
                try{

Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=
DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","abc123");

DatabaseMetaData metadata=con.getMetaData();

System.out.println("Driver Name: "+metadata.getDriverName());
System.out.println("Driver Version: "+metadata.getDriverVersion());
System.out.println("UserName: "+metadata.getUserName());
System.out.println("Database Product Name: "+metadata.getDatabaseProductName());
System.out.println("Database Product Version: "+metadata.getDatabaseProductVersion());

con.close();
}catch(Exception e){ System.out.println(e);}
}
}
```

**Output:**

```
Driver Name: Oracle JDBC Driver
     Driver Version: 10.2.0.1.0XE
     Database Product Name: Oracle
     Database Product Version: Oracle Database 10g Express Edition
                                Release 10.2.0.1.0 -Production
```

**//Example #2: DatabaseMetaData interface that prints total number of tables**

```
import java.sql.*;
pubic class TableDetails{

public static void main(String args[]){

try{
       Class.forName("oracle.jdbc.driver.OracleDriver");
```

```
        Connection con=
DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","abc123");

DatabaseMetaData dbmd = con.getMetaData();

String table[]={"TABLE"};
        //if TABLE is replaced with VIEW, we can get VIEW information

ResultSet rs=dbmd.getTables(null,null,null,table);
while(rs.next()) {
        System.out.println(rs.getString(3));
}

con.close();

}catch(Exception e){ System.out.println(e);}

}
}
```